

国外计算机科学教材系列

# Java大学教程

Java How to Program

■ [美] Harvey M. Deitel 著  
Paul J. Deitel

■ 吴红宇 史晓华 邵 晖 等译  
■ 高仲仪 审校



DEITEL

Premier  
Hall



电子工业出版社  
Publishing House of Electronics Industry  
<http://www.phei.com.cn>

# 经典教材

## Java 大学教程、

Java How to Program

Java 以其强大的多媒体功能、平台无关性、面向对象的编程特点以及基于网络的应用程序和 applet，掀起了软件开发的又一次革命。本书是 Deitel 的系列丛书 “How to Program” 中的经典力作，本丛书也是世界上最广泛使用的大学编程语言的入门教程。这本书将向读者展示 Java 的基本概念和编程技巧，并介绍了 Java 中面向对象编程的相关知识，其中的主要课题包括：

- 应用程序/applet
- 类/对象/接口
- 封装/内部类
- OOP/继承/多态
- 图形/图像/动画
- 异常/多线程
- 文件/流
- 网络/客户/服务器
- 数据结构/集合

### Harvey M. Deitel 博士

Deitel & Associates 公司的执行总裁，有着近 40 年计算机领域的工作经验，进行过大量深入的教学研究，是世界一流的计算机科学教授和研讨会演讲人。他是几十本专著和多媒体软件包的作者或合著者，是全球知名的计算机教材作者。

### Paul J. Deitel

Deitel & Associates 公司的技术总监，是麻省理工学院 Sloan 管理学校的毕业生，主修信息技术。他和 Harvey M. Deitel 博士一起创作了十余本专著和多媒体软件包。

ISBN 7-5053-7693-4



9 787505 376939 >



责任编辑：冯小贝  
封面设计：毛惠庚

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-5053-7693-4/TP · 4444 定价：65.00 元

国外计算机科学教材系列

# Java 大学教程

Java How to Program

[ 美 ] Harvey M. Deitel 著  
Paul J. Deitel

奚红宇 史晓华 邵 晖 等译  
高仲仪 审校

電子工業出版社  
Publishing House of Electronics Industry  
北京 · BEIJING

## 内 容 简 介

本书以初学者为起点,循序渐进地介绍了面向对象的Java编程语言,系统地讨论了Java的基本概念和编程技术。全书共分为18章,首先从基本的Java理论开始,讲解了Java的基本数据类型和控制结构、Java中的方法、数组和字符串,以及基于对象的编程和面向对象的编程。书中还讨论了很多有关Java的高级课题,包括图形、图形用户界面组件、异常处理、多线程、多媒体、文件和流、网络、数据结构以及Java工具包和位处理。全书内容丰富、构思严谨、条理清晰,写作方法别具一格,并且提供了大量实用、有趣的练习,可以使读者在较短的时间内掌握基本的和最新的编程技术。

本书是高等院校进行编程语言和Java教学的教材,也是软件设计人员进行Java程序开发的宝贵参考资料。

Simplified Chinese edition Copyright © 2003 by PEARSON EDUCATION NORTH ASIA LIMITED and Publishing House of Electronics Industry.

Java How to Program, ISBN: 0136325890 by Harvey M. Deitel, Paul J. Deitel, Copyright © 1997.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书中文简体字翻译版由电子工业出版社和Pearson Education培生教育出版北亚洲有限公司合作出版。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有Pearson Education培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记号:图字:01-98-1315

### 图书在版编目(CIP)数据

Java大学教程/(美)戴特尔(Deitel)等著;奚红宇等译.-北京:电子工业出版社,2003.4

(国外计算机科学教材系列)

书名原文:Java How to Program

ISBN 7-5053-7693-4

I. J... II. ①戴... ②奚... III. JAVA语言-程序设计-高等学校-教材 IV. TP312

中国版本图书馆CIP数据核字(2003)第026981号

责任编辑:冯小贝

印刷者:北京兴华印刷厂

出版发行:电子工业出版社 <http://www.phei.com.cn>

北京市海淀区万寿路173信箱 邮编:100036

经 销:各地新华书店

开 本:787×1092 1/16 印张:48.75 字数:1248千字

版 次:2003年4月第1版 2003年4月第1次印刷

定 价:65.00元

凡购买电子工业出版社的图书,如有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系。联系电话:(010)68279077



## 出版说明

21世纪初的5至10年是我国国民经济和社会发展的关键时期,也是信息产业快速发展的关键时期。在我国加入WTO后的今天,培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡,是我国面对国际竞争时成败的关键因素。

当前,正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期,为使我国教育体制与国际化接轨,有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材,以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验,翻译出版了“国外计算机科学教材系列”丛书,这套教材覆盖学科范围广、领域宽、层次多,既有本科专业课程教材,也有研究生课程教材,以适应不同院系、不同专业、不同层次的师生对教材的需求,广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时,我们也适当引进了一些优秀英文原版教材,本着翻译版本和英文原版并重的原则,对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上,我们大都选择国外著名出版公司出版的高校教材,如Pearson Education培生教育出版集团、麦格劳-希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者,如道格拉斯·科默(Douglas E. Comer)、威廉·斯托林斯(William Stallings)、哈维·戴特尔(Harvey M. Deitel)、尤利斯·布莱克(Uyless Black)等。

为确保教材的选题质量和翻译质量,我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士,也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中,为提高教材质量,我们做了大量细致的工作,包括对所选教材进行全面论证;选择编辑时力求达到专业对口;对排版、印制质量进行严格把关。对于英文教材中出现的错误,我们通过与作者联络和网上下载勘误表等方式,逐一进行了修订。

此外,我们还将与国外著名出版公司合作,提供一些教材的教学支持资料,希望能为授课老师提供帮助。今后,我们将继续加强与各高校教师的密切联系,为广大师生引进更多的国外优秀教材和参考书,为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

## 教材出版委员会

主 任	杨芙清	北京大学教授 中国科学院院士 北京大学信息与工程学部主任 北京大学软件工程研究所所长
委 员	王 珊	中国人民大学信息学院院长、教授
	胡道元	清华大学计算机科学与技术系教授 国际信息处理联合会通信系统中国代表
	钟玉琢	清华大学计算机科学与技术系教授 中国计算机学会多媒体专业委员会主任
	谢希仁	中国人民解放军理工大学教授 全军网络技术研究中心主任、博士生导师
	尤晋元	上海交通大学计算机科学与工程系教授 上海分布计算技术中心主任
	施伯乐	上海国际数据库研究中心主任、复旦大学教授 中国计算机学会常务理事、上海市计算机学会理事长
	邹 鹏	国防科学技术大学计算机学院教授、博士生导师 教育部计算机基础课程教学指导委员会副主任委员
	张昆藏	青岛大学信息工程学院教授

## 译 者 序

随着 Internet 和 WWW 的迅猛发展,世界上任意位置的计算机都可以通过无处不在的通信网络进行交流。网络构成了一个全新的“虚拟世界”,而信息则如同潮水一样不断涌出,并呈现出“爆炸性”的增长。

Sun Microsystems 公司从 1991 年开始进行的“Green”研究计划,终于在 1995 年 5 月演变成了 Java 语言。从它诞生的那天起,Java 这种面向对象的编程语言就引起了众多有识之士的共同关注。由于商业领域投入了极大的热情,使得 Java 在短短几年内就可以迅速地传播和发展,并获得了大家的普遍认同。

本书的两位作者均为资深的计算机教材作者,而且他们一直在大学从事计算机编程语言的教学工作,有着丰富的写作和编程经验。本书自出版后就迅速成为全球最畅销的有关 Java 的大学教程,两位作者则将这本书定位为大学低年级学生的编程课程教材。因此,对于一个没有任何计算机编程经验的初学者而言,不但可以通过本书来轻松地学习如何应用 Java 语言进行编程,而且还将了解面向对象编程技术的思想要点,以及经典的结构化编程技术的精髓。

书中还给出了大量的如何高效地编写出高质量程序的技巧与经验,其中包括“编程技巧”、“常见编程错误”、“测试与调试提示”、“性能提示”、“可移植性提示”和“软件工程视点”。这些技巧和经验是十分宝贵的,一定会使读者受益匪浅。

本书主要由奚红宇、史晓华、邵晖、高仲仪组织翻译工作,同时参加翻译工作的还有张莉、杨玉光、纪东颖、赵姝、李逸、程明一、樊同思、付文鹏、马潇兰、苏文启、刘光明、王洋。全书最后由高仲仪负责审校与统稿。由于译者水平有限,书中难免出现疏漏之处,敬请读者批评指正。

## 关于作者

**Harvey M. Deitel** 博士：Deitel & Associates 公司的总裁，有着近 40 年计算机领域的工作经验，是世界著名的计算机科学教员和研讨会演讲人。Deitel 博士拥有麻省理工学院（MIT）的学士、硕士学位以及波士顿大学的博士学位。他曾经在 IBM 和 MIT 的虚拟内存操作系统项目中进行过开拓性的研究工作，开发了如今在 UNIX、Windows NT 和 OS/2 等一些操作系统中广泛实现的技术。Deitel 博士有着 20 多年的大学教学经验，在成立 Deitel & Associates 公司以前，他是波士顿大学计算机科学系的系主任。Deitel 博士是几十本专著和多媒体软件包的著者或合著者，他的著作获得了国际上的广泛认可，并已经翻译成日文、俄文、西班牙文、韩文、法文、波兰文和葡萄牙文等语言。Deitel 博士是全球知名的计算机教材作者。

**Paul J. Deitel**：Deitel & Associates 公司的执行副总裁，是 MIT Sloan 管理学院的毕业生，主修信息技术。通过 Deitel & Associates 公司，他已经为 Digital Equipment Corporation、Sun Microsystems、Rogue Wave Software、Software 2000、Computervision、Stratus、Fidelity、Cambridge Technology Partners、Open Environment Corporation、One Wave、Hyperion Software、Lucent Technologies、Adra Systems、Entergy、CableData Systems、NASA、National Severe Storm Laboratory 和 IBM 等多家公司讲授 Java、C 和 C++ 语言；并且参与了 Deitel & Associates 公司、Prentice Hall 和 Technology Education Network 的一个合作计划，通过卫星教授 Java 课程。Paul J. Deitel 与 Harvey M. Deitel 博士一起完成了 10 余本专著和多媒体软件包。

Deitel 一家是世界上最畅销的大学编程语言入门教材的著者或合著者，这套教材包括：“C How to Program”、“C++ How to Program”和“Visual Basic 6 Program”等。他们也是 Prentice Hall 的多媒体教材“C & C++ Multimedia Cyber Classroom”、“Java Multimedia Cyber Classroom”以及“Visual Basic 6 Multimedia Cyber Classroom”的合作开发者。

# 前 言

欢迎来到Java以及令人激动的Internet、Intranet和WWW的编程世界！本书是由一老一少编写而成的。老者（Harvey M. Deitel）有着近40年的编程和教学经验；少者（Paul J. Deitel）从事编程工作已近20年，并有着丰富的教学与写作经历。老者经验丰富，少者精力充沛；老者追求清晰明确，少者力求丰富的表现；老者喜欢优雅与美观，少者则追求快速得到结果。一老一少的合作，使本书的信息更丰富、内容更有趣。

## 为什么要编写本书

20多年来，Harvey M. Deitel博士在大学讲授编程课程时，不断强调的是结构明确、书写清晰的程序。在这些课程中，尤其强调如何高效地使用程序的控制结构和功能模块。

Deitel博士在大学教授这些课程的经验 and 体会将充分展现在本书中。我们的授课经验是，学生在学习前几章关于控制结构和方法（Java中关于函数的术语）的课程时，他们所用的方式同学习传统的Pascal或C编程语言时是一致的。但是仍然有一点不同，学生们充分地认识到他们正在学习一种最前沿的编程语言（Java）以及一个最前沿的语言范例（面向对象的编程）；并且知道一旦离开大学环境进入Internet这个无处不在的世界时，这些知识是多么重要。这一点增加了学生们对课程的学习热情，这对教师来说很有帮助，它使教师认识到必须将基本的语言同实际的类库相结合，才能够满足学生们不断增长的渴求知识的愿望，学生们将会发现，他们可以利用Java完成许多出色的工作，因此他们愿意付出额外的努力。

我们的目标是很明确的：编写一本用于Java语言编程的大学教程，学生也许在编程方面很少有或是完全没有经验，但是在课程的深度和内容的严密程度上，这一教程可以同传统的C语言、C++语言的教学要求相适应。为了这一目标，我们的教材尽量做到内容丰富，其中包含编程语言控制结构的基本原理、面向对象的编程、Java语言以及Java的类库。本书是第一本讲述Java语言的大学教程。由于过去我们编写的几本书，例如“C How to Program”以及“C++ How to Program”，全都是它们所在领域的世界范围内的畅销教材，因此我们及时推出了有关Java的这本经典教程。

我们对Java充满信心。它的创造者——Sun Microsystems公司对Java的定位相当明确：首先这种新的语言基于两种全球使用最广的语言：C和C++。这使得Java一出现就能覆盖一个巨大的潜在使用群，即那些当前在全球大多数新兴的操作系统、通信系统、数据库系统和个人电脑应用系统中担当重要角色的熟练程序员。其次，Java摒弃了大量较为复杂、易于出错的语言特性（例如指针、模板、运算符重载和多继承等）。通过删除仅在很少的特定领域中使用的较为特殊的一些语言特性，Java语言变得相当精练。Java拥有真正的可移植性，这使得它可以更好地应用在Internet及WWW上。那些人们真正需要的特性，诸如字符串、图形、用户图形界面、异常处理、多线程、多媒体（音频、图像、动画——最终是视频）、文件处理、基于Internet的客户/服务器结构以及数据包预处理等，Java都会将其提供给用户。最后，Sun Microsystems公司免费将Java语言的基本环境提供给全球数以百万计的潜在用户。

Java起初是为了给WWW主页增加一些“动感”而进行设计的。不同于以往只有文本和静态图

形的网络主页，现在的主页依靠音频、动画、视频和三维图像而“活动起来”。但这不仅仅是为了个人喜好，这些特性正是商业机构和各种组织为了满足信息产业的日益发展所必需的。因此，我们可以立即预见到Java的巨大潜能，Java将成为全球最重要的通用编程语言之一。

市面上有一些已经出售的Java产品。当然，读者没有必要在阅读本书时就购买并使用它们，因为我们在撰写时采用的是Sun Microsystems公司在Internet上提供的免费软件。我们已经发现，许多市面上的Java产品对于商业及个人用户都具有很强的吸引力。

Java激发着人们和各种组织发掘他们的创造力，这在我们开设的Java培训班中就已经发现。一旦进入实验室，学生们就变得流连忘返。他们是那样地渴望获取经验，甚至包括我们尚未讲授的各种Java类库的内容。学生们设计的应用程序远远超过了我们曾经在各个C及C++培训班中力图让学生们编写的。

坦率地说，Java本身也存在缺点，但是Java是一种新兴的语言——时间将使它成熟起来。我们对Java的兴趣是巨大的，人们的确喜欢采用“Java模式”进行工作。Java现有的缺点已经得到关注，并且可以在将来得到解决。例如，无论组织还是个人都希望在Internet上开展业务。Internet是个缺乏基本安全性的通信媒介。但是在将现有的技术进行改造后，我们可以在不安全的通道上进行有安全保证的传输——虽然这看上去相互矛盾并且短期内无法实现。

计算机世界从未经历过像今天Internet/WWW/Java这样的“爆炸性”发展。人们渴望相互交流，人们需要相互交流。虽然在文明产生时，人类交流和通信就开始了，但是直到今天，计算机通信仍然局限在数字、可见字符和各种特殊字符上。下一次浪潮必然是多媒体。人们希望传输图片并且希望这些图片是彩色的。人们希望传输语音、声音和音频剪辑，希望传输动态的彩色视频。从某种程度上来说，我们的确需要三维的、动态的图像。现在的二维显示设备最终将被三维设备所取代，这使得我们可以在家中感受到“家庭影院”的效果，我们的起居室将成为一个小型体育馆。我们可以同远在地球另一边的商业伙伴一起参加网络视频会议，大家虽然相隔千里却感觉近在咫尺。这些可能性就在眼前，Java是使梦想成真的关键角色。

有人断言，Internet将完全取代电话系统。那么，为什么不继续设想下去？Internet将最终代替我们今天使用的广播和电视系统。不难想像，Internet将完全以电子报刊的形式取代现今的报纸。我们现在所阅读的书籍有朝一日将同收音机、电视机和报纸一同出现在博物馆的“古代文明媒体”的展览中。

## 教学方法

本书包含丰富的示例、练习以及在许多领域开发的项目，这些都给学生们提供了一个解决他们感兴趣的实际问题的机会。本书关心的是怎样符合良好的软件工程原则，并且将重点放在如何表述清晰的程序。我们愿意使用实例而不是晦涩的术语和严格的语法描述来讲述语言本身。书中的每一个例子都已经在几个Java平台上经过测试，这些平台包括Sun Solaris、Microsoft Windows 95和Windows NT。

### 面向对象的技术和Java applet

本书从第1章就开始学习面向对象的编程、Java applet（小程序）以及基本的图形用户界面设计。有人曾经提醒我们，这是一个过于快速的教学方式，但是学习这门课程的学生们却真正渴望了解这些新的知识。Java有很丰富的内容等待着我们去学习，为什么不马上进行！无论怎样，Java都非同寻常，它充满了乐趣并且学生们可以立即看到学习成果。利用Java附带的类库（可重用的模块），学生们可以迅速使他们的图形、动画、基于多媒体的音频、多线程以及基于网络的程序运行

起来,他们可能完成令人印象深刻的项目。这些学生可能比那些学习C/C++编程语言的同学更富有创造性并且编程效率更高。

## 活动代码教学法

本书是同活动代码的例子相结合的。事实上,每一个新的概念都是通过一个完整的、可工作的Java程序(Java applet或Java应用程序)表达的,并通过一个窗口立即显示程序的输出结果。我们将这种教学及写作方法称为活动代码(Live-Code)教学方法。我们通过使用这门语言来教授这门语言。在我们阅读这些程序时,更像在机器上输入并执行它们。

## 学习目标

每一章的开头都列出了一个“学习目标”,它告诉学生这一章的学习要求是什么,同时也向学生提供了一个机会,使得他们在学习完一章后能够判断自己是否达到了预定的目标。

## 12 087行代码、200个程序示例(包含程序输出)

我们通过文中完整的、可执行的Java程序来展现Java的特点。这是我们教授课程及撰写教材时的重点。我们称其为“活动代码教学法”。其中一些程序带有抓屏结果,显示了执行完这个程序后的输出。这使学生能确认程序的实际输出结果同期望的结果是否一致。仔细地阅读本书中的例子,与在机器上输入、执行这些程序的效果差不多。这些例子既有包含几行的小程序,同时也有几百行的、较为重要的应用程序。学生们在学习本书的同时,可以在自己的机器上执行相应的程序,并进行适当的修改和调试。

## 166个图表

本书包含大量的表格、插图以及程序的输出。例如在讨论控制结构的时候,书中便给出了精心绘制的流程图。注意,我们并没有将使用流程图作为程序开发的手段,但是为了详细说明Java的控制结构,仍然使用了这种简洁的流程表示方法。

## 369个编程技巧

我们在书中使用了大量的编程技巧,帮助学生们将注意力集中在程序开发的重要方面。我们强调这些技巧,并将数百个技巧分成以下几类:“编程技巧”,“常见编程错误”,“测试与调试提示”,“性能提示”,“可移植性提示”以及“软件工程视点”。这些技巧总结了我们几十年的编程和教学经验。一位已成为数学专家的学生最近告诉我们,她感觉是这些方法好像数学教材中强调的公理、定理和推论,提供了建立良好的软件工程的基础。

### 62个编程技巧

当讲授预备性课程时,我们介绍编程的一个原则就是“清晰性”。我们告诉学生,将在这些“编程技巧”中强调使程序更清晰、更易懂和更易于理解的技术。

### 139个常见编程错误

学习一种语言常常容易犯这样或那样的错误。让学生们注意这些“常见编程错误”有助于避免重犯相同的错误。这同样可以帮助读者在今后工作中少犯错误。

### 29个测试与调试提示

最初设计这部分提示时,我们希望利用它来正确告诉人们如何测试及调试Java程序。实际上,同C和C++比较起来,我们更多地介绍了那些能够减少可能的“错误”并且简化测试和调试步骤的技巧及经验。

### 50 个性能提示

根据我们的经验,在最初的编程课程上教会学生编写清晰、易于理解的程序是最重要的。但学生们希望编写运行最快、内存需求更小、操作更简化的程序。学生们非常关心程序的操作性能,希望知道怎样补充和调试程序,所以我们给出了这 50 个“性能提示”,从而为读者提供可以改善程序性能的方法。

### 8 个可移植性提示

这些提示帮助学生编写可移植的代码,并且深入到 Java 内部,以便了解 Java 是如何实现高度可移植性的。在“C How to Program”一书中给出很多“可移植性提示”,但本书则要少得多。这是因为 Java 在设计时就遵循了自顶向下的可移植设计,因此对于 Java 程序员来说,所要考虑的可移植性问题比 C 及 C++ 的程序员少得多。

### 81 个软件工程视点

面向对象的编程方式要求我们重新考虑传统的编程方法。Java 是实现良好软件工程的一种高效的编程语言。在软件系统尤其是大型系统中,“软件工程视点”强调了系统的结构和设计方案对整体构造的影响。本书的许多知识对学习高级课程也是很有帮助的,当学生们从事大型、复杂系统的设计工作时,这些经验就变得相当重要。

## 小结

每一章的结束都带有附加的教学方法,提供了本章完整的项目符号形式的总结。平均每章有 34 个小结项。这些小结有助于学生们复习本章内容并加强理解关键的概念。

## 术语

每一章都提供了按字母顺序列出的本章的重要术语,以便再一次强调这些基本内容。平均每章有 87 个术语。

## 346 个自测练习和答案

附加的自测练习和答案用于自学,可以使学生有机会利用这些题目建立信心,并为以后的学习做好准备。应该尽量完成所有的练习并检查练习结果。

## 877 个练习

每一章均提供了大量的练习,包括复习重要术语和概念的题目,编写简单的 Java 语句,编写 Java 的方法和类中的一部分;编写完整的 Java 方法、类、applet 和应用程序;完成主要的学期项目。这些练习的范围广泛,允许教师根据学生的需要来调整课程。也可以将这些练习作为家庭作业、小测验和考试题目。

## 参考文献

对于需要进一步学习的读者可以参看附加的参考文献。



# 目 录

第 1 章 计算机和 Java applet 简介 .....	1
1.1 简介 .....	1
1.2 什么是计算机 .....	4
1.3 计算机的组织结构 .....	4
1.4 操作系统的发展 .....	5
1.5 个人计算、分布式计算和客户 / 服务器计算 .....	5
1.6 机器语言、汇编语言和高级语言 .....	6
1.7 C++ 的历史 .....	7
1.8 Java 的历史 .....	8
1.9 Java 的类库 .....	8
1.10 其他高级语言 .....	9
1.11 结构化编程 .....	9
1.12 一个典型 Java 环境的基础知识 .....	9
1.13 预览本书 .....	11
1.14 关于 Java 和本书的一般注意事项 .....	13
1.15 Java 编程介绍 .....	15
1.16 一个简单的例子：打印一行文本 .....	15
1.17 另一个 Java 程序：整数相加 .....	19
1.18 关于内存的概念 .....	24
1.19 算术运算 .....	25
1.20 条件判断：相等运算符和关系运算符 .....	28
小结 .....	32
术语 .....	35
自测练习 .....	37
自测练习答案 .....	38
练习 .....	39
第 2 章 控制结构（一） .....	43
2.1 简介 .....	43
2.2 算法 .....	43
2.3 伪代码 .....	43
2.4 控制结构 .....	44
2.5 if 选择结构 .....	46
2.6 if / else 选择结构 .....	47
2.7 while 循环结构 .....	51

2.8 构造算法: 实例1 (计数器控制循环).....	52
2.9 自顶向下、逐步求精的构造算法: 实例2 (标志控制循环).....	56
2.10 自顶向下、逐步求精的构造算法——实例3 (嵌套的控制结构).....	61
2.11 赋值运算符.....	65
2.12 自增和自减运算符.....	66
2.13 基本数据类型.....	68
2.14 常见的转义序列.....	69
小结.....	70
术语.....	71
自测练习.....	72
自测练习答案.....	73
练习.....	74
<b>第3章 控制结构 (二)</b> .....	80
3.1 简介.....	80
3.2 计数器控制循环的实质.....	80
3.3 for 循环结构.....	82
3.4 使用 for 结构的例子.....	85
3.5 switch 多重选择结构.....	88
3.6 do/while 循环结构.....	91
3.7 break 和 continue 语句.....	93
3.8 带标记的 break 和 continue 语句.....	95
3.9 逻辑运算符.....	97
3.10 结构化编程小结.....	101
小结.....	105
术语.....	105
自测练习.....	106
自测练习答案.....	107
练习.....	108
<b>第4章 方法</b> .....	112
4.1 简介.....	112
4.2 Java 中的程序模块.....	112
4.3 Math 类的方法.....	113
4.4 方法.....	114
4.5 方法定义.....	115
4.6 参数类型提升.....	119
4.7 Java API 软件包.....	120
4.8 生成随机数.....	121
4.9 案例: 一个机会游戏.....	124
4.10 自动变量.....	128
4.11 作用域规则.....	129

4.12 递归 .....	131
4.13 使用递归的例子: 斐波纳契数列 .....	133
4.14 递归与迭代 .....	136
4.15 方法重载 .....	138
4.16 Applet 类的方法 .....	140
小结 .....	141
术语 .....	143
自测练习 .....	144
自测练习答案 .....	146
练习 .....	149
<b>第5章 数组 .....</b>	<b>157</b>
5.1 简介 .....	157
5.2 数组 .....	157
5.3 声明数组和分配数组 .....	159
5.4 使用数组的实例 .....	160
5.4.1 分配数组并初始化数组元素 .....	160
5.4.2 使用初始化值列表来初始化数组元素 .....	161
5.4.3 计算存储在数组元素中的值 .....	162
5.4.4 对数组元素求和 .....	164
5.4.5 使用数组分析调查结果 .....	165
5.5 引用和引用参数 .....	169
5.6 向方法传递数组 .....	170
5.7 数组排序 .....	172
5.8 数组查找: 线性查找和二分查找 .....	174
5.8.1 线性查找 .....	174
5.8.2 二分查找 .....	175
5.9 多维数组 .....	178
小结 .....	184
术语 .....	185
自测练习 .....	185
自测练习答案 .....	186
练习 .....	187
递归练习 .....	195
特殊小节: 建立自己的计算机 .....	198
<b>第6章 基于对象的编程 .....</b>	<b>204</b>
6.1 简介 .....	204
6.2 通过类实现一个抽象数据类型 Time .....	205
6.3 类作用域 .....	208
6.4 控制对成员的访问 .....	209
6.5 实用方法 .....	210

6.6 初始化类对象：构造函数 .....	213
6.7 使用重载的构造函数 .....	214
6.8 使用 set 和 get 方法 .....	217
6.9 软件可重用性 .....	221
6.10 final 实例变量 .....	222
6.11 复合：作为其他类的实例变量的对象 .....	223
6.12 软件包访问 .....	226
6.13 使用 this 引用 .....	227
6.14 终止函数 .....	231
6.15 静态类成员 .....	231
6.16 数据抽象和信息隐藏 .....	234
6.16.1 案例：队列抽象数据类型 .....	236
小结 .....	236
术语 .....	237
自测练习 .....	238
自测练习答案 .....	238
练习 .....	238
<b>第7章 面向对象的编程</b> .....	242
7.1 简介 .....	242
7.2 超类和子类 .....	243
7.3 protected 成员 .....	245
7.4 超类对象和子类对象之间的关系 .....	245
7.5 在子类中使用构造函数和终止函数 .....	249
7.6 从子类对象到超类对象的隐式转换 .....	252
7.7 使用继承的软件工程 .....	253
7.8 复合与继承 .....	254
7.9 案例分析：点、圆、圆柱体 .....	254
7.10 多态简介 .....	259
7.11 类型域和 switch 语句 .....	259
7.12 动态方法绑定 .....	259
7.13 final 方法和类 .....	260
7.14 抽象超类和具体类 .....	260
7.15 多态的例子 .....	261
7.16 案例分析：一个使用多态的工资支付系统 .....	262
7.17 新类和动态绑定 .....	268
7.18 案例分析：继承接口与实现 .....	269
7.19 基本类型的类型包装类 .....	273
小结 .....	274
术语 .....	275
自测练习 .....	276

自测练习答案 .....	277
练习 .....	277
<b>第8章 字符串和字符 .....</b>	<b>279</b>
8.1 简介 .....	279
8.2 字符和字符串的基础 .....	279
8.3 String 构造函数 .....	280
8.4 String 方法: length、charAt、getChars、getBytes .....	282
8.5 比较 String .....	284
8.6 String 方法 hashCode .....	289
8.7 在 String 中定位字符和子字符串 .....	290
8.8 从 String 中提取子字符串 .....	292
8.9 连接 String .....	293
8.10 其他的 String 方法 .....	294
8.11 使用 String 方法 valueOf .....	296
8.12 String 方法 intern .....	297
8.13 StringBuffer 类 .....	299
8.14 StringBuffer 构造函数 .....	300
8.15 StringBuffer 的 length、capacity、setLength 和 ensureCapacity 方法 .....	301
8.16 StringBuffer 的 charAt、setCharAt 和 getChars 方法 .....	302
8.17 StringBuffer 的 append 方法 .....	304
8.18 StringBuffer 的 insert 方法 .....	305
8.19 Character 类的例子 .....	307
8.20 StringTokenizer 类 .....	313
8.21 洗牌和发牌的模拟 .....	315
小结 .....	318
术语 .....	320
自测练习 .....	322
自测练习答案 .....	322
练习 .....	322
特殊小节: 高级字符串操作练习 .....	324
挑战性的字符串操作项目 .....	327
<b>第9章 图形 .....</b>	<b>328</b>
9.1 简介 .....	328
9.2 图形环境和图形对象 .....	329
9.3 绘制字符串、字符和字节 .....	330
9.4 颜色控制 .....	331
9.5 字体控制 .....	336
9.6 绘制线条 .....	343
9.7 绘制矩形 .....	344
9.8 绘制圆角矩形 .....	345

9.9 绘制三维矩形 .....	347
9.10 绘制椭圆 .....	349
9.11 绘制圆弧 .....	350
9.12 绘制多边形 .....	352
9.13 屏幕操作 .....	355
9.14 绘图模式 .....	356
小结 .....	358
术语 .....	360
自测练习 .....	361
自测练习答案 .....	362
练习 .....	362
<b>第 10 章 图形用户界面组件 (一)</b> .....	365
10.1 简介 .....	365
10.2 标签 .....	366
10.3 掀压式按钮 .....	369
10.4 文本字段 .....	372
10.5 选择按钮 .....	376
10.6 复选框按钮和单选按钮 .....	378
10.7 列表 .....	382
10.8 面板 .....	386
10.9 鼠标事件 .....	387
10.10 键盘事件 .....	394
10.11 FlowLayout 布局管理器 .....	399
10.12 BorderLayout 布局管理器 .....	404
10.13 GridLayout 布局管理器 .....	407
小结 .....	409
术语 .....	411
自测练习 .....	413
自测练习答案 .....	414
练习 .....	414
<b>第 11 章 图形用户界面组件 (二)</b> .....	419
11.1 简介 .....	419
11.2 文本区域 .....	419
11.3 画板 .....	422
11.4 滚动条 .....	426
11.5 定制组件 .....	430
11.6 框架 .....	432
11.7 菜单 .....	441
11.8 对话框 .....	448
11.9 高级布局管理器 .....	455

11.10 CardLayout 布局管理器 .....	455
11.11 GridBagLayout 布局管理器 .....	458
11.12 不使用布局管理器 .....	465
11.13 程序员自定义的布局管理器 .....	466
小结 .....	470
术语 .....	471
自测练习 .....	473
自测练习答案 .....	474
练习 .....	474
<b>第 12 章 异常处理</b> .....	<b>477</b>
12.1 简介 .....	477
12.2 何时使用异常处理 .....	479
12.3 其他的错误处理技术 .....	479
12.4 Java 异常处理的基础 .....	480
12.5 一个异常处理的简单实例：除数为零 .....	480
12.6 try 程序块 .....	483
12.7 抛出异常 .....	484
12.8 捕获异常 .....	484
12.9 重抛出异常 .....	486
12.10 throws 子句 .....	486
12.11 构造函数、终止函数和异常处理 .....	490
12.12 异常和继承 .....	491
12.13 finally 程序块 .....	491
12.14 使用 printStackTrace 和 getMessage 方法 .....	495
小结 .....	497
术语 .....	498
自测练习 .....	499
自测练习答案 .....	499
练习 .....	500
<b>第 13 章 多线程</b> .....	<b>502</b>
13.1 简介 .....	502
13.2 Thread 类：线程方法介绍 .....	504
13.3 线程状态：一个线程的生命周期 .....	505
13.4 线程优先级与线程调度 .....	506
13.5 线程同步 .....	509
13.6 未使用线程同步的生产者/消费者关系 .....	510
13.7 使用线程同步的生产者/消费者关系 .....	513
13.8 生产者/消费者的关系：循环缓冲区 .....	516
13.9 精灵线程 .....	521
13.10 Runnable 接口 .....	521

13.11 线程组 .....	524
小结 .....	526
术语 .....	528
自测练习 .....	529
自测练习答案 .....	530
练习 .....	530
<b>第 14 章 多媒体：图像、动画和声音 .....</b>	<b>534</b>
14.1 简介 .....	534
14.2 加载、显示和按比例调整图像 .....	535
14.3 动画介绍：图像的循环 .....	537
14.4 图形双缓存 .....	539
14.5 利用 MediaTracker 来监视图像的加载 .....	542
14.6 利用一个独立线程来运行动画 .....	545
14.7 加载和播放音频剪辑 .....	548
14.8 通过 HTML 的 param 标记来定制 applet .....	550
14.9 图像映射 .....	554
小结 .....	557
术语 .....	559
自测练习 .....	559
自测练习答案 .....	560
练习 .....	560
<b>第 15 章 文件和流 .....</b>	<b>566</b>
15.1 简介 .....	566
15.2 数据组织 .....	566
15.3 文件和流 .....	568
15.4 创建顺序访问文件 .....	571
15.5 从顺序访问文件中读取数据 .....	576
15.6 更新顺序访问文件 .....	584
15.7 随机访问文件 .....	584
15.8 创建随机访问文件 .....	585
15.9 向随机访问文件中随机地写入数据 .....	588
15.10 从随机访问文件中顺序地读取数据 .....	592
15.11 案例：交易处理程序 .....	595
15.12 File 类 .....	605
15.13 对象的输入/输出 .....	608
小结 .....	609
术语 .....	611
自测练习 .....	613
自测练习答案 .....	614
练习 .....	615



<b>第 16 章 网络</b>	618
16.1 简介	618
16.2 利用 URL	619
16.3 采用 URL 的流连接从服务器上读取文件	621
16.4 建立一个简单的服务器 (采用流套接字)	623
16.5 建立一个简单的客户 (采用流套接字)	624
16.6 通过流套接字进行的客户 / 服务器交互	625
16.7 采用数据报方式进行无连接的客户 / 服务器交互	629
16.8 采用多线程服务器实现的客户 / 服务器间的三连棋游戏	635
16.9 网络 and 安全性	644
小结	645
术语	646
自测练习	647
自测练习答案	648
练习	649
<b>第 17 章 数据结构</b>	651
17.1 简介	651
17.2 自引用的类	651
17.3 动态内存请求	652
17.4 链表	653
17.5 堆栈	661
17.6 队列	664
17.7 树	666
小结	671
术语	671
自测练习	672
自测练习答案	673
练习	674
特殊小节: 建立自己的编译器	679
<b>第 18 章 Java 工具包和位处理</b>	691
18.1 简介	691
18.2 Vector 类和 Enumeration 接口	691
18.3 Stack 类	696
18.4 Dictionary 类	699
18.5 Hashtable 类	699
18.6 Date 类	704
18.7 Observable 类和 Observer 接口	705
18.8 Properties 类	708
18.9 Random 类	711
18.10 位处理和位运算符	712

18.11 BitSet 类 .....	720
小结 .....	723
术语 .....	726
自测练习 .....	728
自测练习答案 .....	729
练习 .....	729
附录 A 运算符优先级表 .....	732
附录 B ASCII 字符集 .....	734
附录 C 数值系统 .....	735
附录 D 面向对象的电梯模拟程序 .....	745
参考文献 .....	756

# 第1章 计算机和Java applet简介

## 教学目标

- 理解计算机科学的基本概念
- 熟悉不同种类的程序设计语言
- 理解Java的程序开发环境
- 能够编写简单的Java程序
- 能够使用简单的输入/输出语句
- 熟悉基本的数据类型
- 能够使用算术运算符
- 能够编写简单的条件分支语句

## 1.1 简介

欢迎使用Java语言！我们努力为读者创建一个信息丰富、充满乐趣并且富有挑战性的学习环境。Java是一种功能强大的程序设计语言，它不仅对初学者来说充满乐趣，而且对于那些正在建立重要信息系统的高级程序员也具有吸引力。本书正是献给各方面读者的一个行之有效的学习工具。

如何使一本书的内容满足不同类型读者的需求呢？那就是贯穿全书的中心思想要强调如何写出清晰的程序，而无论我们采用的是结构化编程方法还是面向对象的程序设计方法。没有哪个程序员一开始就能熟练编程，我们在书中尽量采用清晰、直接的表达方式。此外，书中还包含大量的图例。也许最重要的是，本书包含大量可工作的Java程序，并且给出了这些程序在机器上运行时的输出结果。

本书的前几章介绍了计算机、程序设计和Java语言的基础知识。学完这一课程的初学者告诉我们，这几章的内容为以后深入了解Java打下了坚实的基础。高级程序员通常快速地浏览前几章，并会发现后面章节中有关Java语言的描述非常严密并富有挑战性。

许多高级程序员告诉我们，他们非常欣赏书中关于结构化编程的处理方法。通常，这些程序员已经在使用C或者Pascal进行编程，但是由于从未系统地学习过结构化程序设计，因此他们的程序并非具有最佳的代码结构。当复习“控制结构”一章中关于结构化编程的内容之后，程序员的C或者Pascal的编程技巧也可以得到提高。所以，不论你是个新手还是经验丰富的高级程序员，这里都有许多东西等着你去学习。

大多数人都喜欢计算机创造的丰富多彩的世界。通过这本书，我们将学会如何命令计算机去进行这些创造。软件（即用来命令计算机去执行动作和进行判断的指令集）控制着计算机，而Java正是当今世界最流行的软件开发语言之一。Java是Sun Microsystems公司开发的，并且可以从WWW（万维网）站点 <http://www.java.sun.com> 上免费得到。

几乎每一个领域都在努力推广使用计算机。由于计算机硬件和软件都在快速地发展，因此尽管

整个商品市场的价格都在上涨,但是计算机的价格却在持续下降。那些20年前价值百万、足足可以占满一个房间的巨型计算机,现在可以做成比指甲还要小的硅芯片,而它的价格也就值几美元。具有讽刺意味的是,硅却是地球上最丰富的资源之一——它只是沙子的一种成分。硅芯片技术使得计算技术如此经济,以至于全球有两亿台通用计算机在商业、工业、政府部门和家庭中广泛应用。几年之后这个数字将再翻一番。

有种种原因使得本书对于读者来说是个挑战。首先,过去几年中程序员可能将C或者Pascal作为他们的第一个程序设计语言。他们学习的编程方法称为结构化程序设计。但是我们将同时学习结构化程序设计和一种新的方法——面向对象的程序设计。为什么要教授两种方法呢?我们确信,面向对象的技术将成为20世纪90年代中期起最重要的程序设计方法。所以,读者在本书中要创建许多对象并针对这些对象进行工作。当然,有时结构化程序设计是实现一个对象内部逻辑的最佳方法。

其次,今后大量基于C的系统(主要采用结构化程序设计方法)将移植为基于C++和Java的系统(主要采用面向对象的程序设计方法)。这里有许多“C语言遗留代码”,因为C语言已经使用了四分之一世纪,并且最近几年的用户迅速增长。一旦人们接触到C++或/和Java,他们就会发现这两种语言要比C强大得多。这使得他们今后将选择C++或/和Java作为所使用的语言。显而易见,用户需要开始将原有系统转换过来。这样,人们开始采用C++或/和Java的面向对象的编程方法,并最终了解到这两种语言的全部优点。

Java必将成为实现基于Internet的应用程序的首选语言。为什么Java有如此之大的吸引力?这里有许多原因。我们可能发现,每个原因都使得Java具有一定的吸引力,但还不足以使大家从那些主要的、得到业界认可的、像C或C++那样的语言转变过来。但综合考虑所有的理由,在推动程序设计语言的教学方面,特别是在介绍和相互交流的层次上,Java提供了一些非常吸引人的功能。最近一个时期,C尤其是C++成为程序员们用来实现大型、复杂系统的重要程序设计语言。但是现在,Java由于可以使用户的主页“活动起来”而将成为大多数人使用的一种语言。因为Java的出现,人们发现它可以实现一个传统的程序设计语言所能达到的大多数目标。

许多年来,C和C++这样的语言由于它们具有良好的可移植性而被许多大学所采用。只要存在一个C/C++的编译器而无论是何种硬件和软件平台,这种语言的教学就可以开展起来。

但是,程序设计世界变得越来越复杂,要求越来越高。今天的用户要求拥有图形用户界面(GUI)的程序。他们要求程序具有多媒体的功能,能够使用图形、图像、动画、音频甚至视频。要求程序可以在像Internet这样的计算机网络上运行,并且可以和网络上的其他应用进行通信;要求程序可以利用多线程的良好特性和复杂性;希望程序的文件处理能力同C/C++语言开发出的程序一样出色。他们希望这些程序不仅仅在自己的桌面或一些较小的网络上应用,而是可以集成为Internet的一部分。他们要求这些程序能够正确而快速地编写出来,并且应用我们常说的“重用、重用、再重用”。他们希望可以利用那些不断增长的重用软件部件。程序员们最终希望这些程序拥有良好的可移植性,这样就可以毫不费力地在另一个平台上运行这些程序而无需修改。Java真正拥有所有的这些特性。

Java对于大学课程具有吸引力的另一个原因在于它是完全面向对象的语言。C++迅速得到广泛应用的一个原因是它对C语言扩充了面向对象的特性。对于数目庞大的C程序员来说,这是一个巨大的进步。C++在包容ANSI/ISO C的同时扩充了面向对象的编程能力。C在1969年就已经出现,从那时起,人们已经编写了数量极为庞大的C代码。因为C++包含C,许多组织发现采用C++是一个很好的升级步骤。在他们的程序员策划面向对象的程序设计时,原有的C代码只需进行一些细小的修改便能在C++的编译器中进行编译,并且程序员们可以继续编写那些像C的C++代码。在时间和条件许可的前提下,程序员最终将C代码的一部分移植到C++。而新的系统就可以完全采用C++中面向对象的技术。这样的策略最初对于许多组织很有吸引力。当然,负而影响也是有的,经过许多

年之后,采用这种策略的组织仍然在生产C风格的C++代码。显然,这表明他们没有立即体会到面向对象程序设计所能带来的好处。许多组织希望他们能够百分之百地采用面向对象的开发方法,但是数量巨大的已有代码妨碍了他们的处理工作。

Java是一种完全面向对象的程序设计语言。用户不可能在Java中编写出C风格的函数,必须创建并利用对象。对于大学教学来说这是一个很有吸引力的特点,学生从一开始就学习面向对象的编程方法,他们一开始就学会使用面向对象的方式进行思考。

当然,这种方式也有不利的地方。那就是使用Java的组织无法在新的程序中继承他们原有的代码。不过,Java提供了一种称为“方言”的功能。这样,已有的C或C++的代码就可以和Java代码集成在一起。虽然这一点看起来有些落后(现在看来正是如此),但是它为大多数团体所面临的问题提供了一个实用的解决方案。Java现在可以从Sun Microsystems公司免费得到。任何人都可以从该公司的站点 <http://www.java.sun.com> 上下载一个完整的Java开发工具箱(JDK)。对于那些需要紧缩预算或希望简化财务申请手续的大学,免费使用Java颇具吸引力。况且,由于改正了错误的版本和新版本的Java可以从Internet上立即得到,因而学校可以随时获得最新的Java软件。

Java可以作为程序设计初学者的首选语言。在撰写本书之前,Deitel & Associates公司为学生们开办了许多Java培训班,他们当中不乏非程序员。我们发现非程序员学习Java要比学习C或C++快得多。他们热切地渴望使用Java各种强大的特性——图形、图形用户界面、多媒体、动画、多线程、网络,他们甚至学完第一次课程就能够成功地写出复杂的Java程序。

多年以来,Pascal一直作为初学者的程序设计入门课程的首选语言。许多人都说C语言对于初学者来说太难了。1992年,我们出版了“C How to Program”一书的第一个版本,目的是说服大学采用C而不是Pascal作为程序设计课程的第一种语言。我们完全采用几十年以来在大学教授程序设计的教学方法,不同的是这一次使用的是C而不是Pascal。我们发现,学生能够像掌握Pascal一样掌握C语言。但仍然有一点明显的不同:由于意识到C语言在今后的实际工业环境中将非常有用,因此学生们倾注了更大的热情。

对于业界的客户而言,他们需要那种能够立即投入到重要项目工作中的熟悉C的毕业生,而不是首先还需要进行培训的学生。

第一版“C How to Program”包含一个60页的介绍C++和面向对象编程技术的部分。我们预见到C++将会得到广泛使用,但那时我们认为在最近几年中,大学还不会将C++和面向对象的编程作为程序设计初学者学习的内容。

1993年,C++和面向对象编程引起了世人的广泛兴趣。但是我们仍然认为大学教学不必立即大规模地转向C++和面向对象编程。因此,1994年1月我们仅出版了篇幅300页的介绍C++和面向对象编程的“C How to Program”第二版。1994年5月,“C++ How to Program”出版了。这本书被许多想在程序设计语言教学中走到前列的大学用做程序设计的入门教程。这些学校对初学者一开始就讲授C++和面向对象的编程技术。

1995年,我们谨慎地开始接触Java。1995年的11月我们在波士顿参加了一个Internet的会议,席中一位来自Sun Microsystems公司的代表在Hynes会议中心为大家介绍了Java。通过他的介绍,我们清醒地意识到Java将成为开发交互的、拥有多媒体功能的主页的重要角色。同时我们也立刻预见到这种语言的巨大潜力。在今天这个充满了图形、图像、动画、音频、视频、网络、多线程以及协作分工的计算机世界中,Java正是许多大学可以用来进行程序设计入门教学的合适语言。因此,我们衷心地希望将本书奉献给那些打算在程序设计入门教学中讲述Java的大学使用。

第1章分成两部分。第一部分介绍了计算机以及计算机编程的基本知识。第二部分直接开始教会读者编写一些简单的Java程序。

做好准备,我们即将开始一个充满挑战和机会的旅程。在学习的同时,请读者将有关本书的有益观点通过电子邮件([deitel@deitel.com](mailto:deitel@deitel.com))发送给我们,我们将尽快地回复。Prentice Hall 公司提供了一个 WWW 站点 <http://www.prenhall.com/deitel>, 其中包含我们为 Prentice Hall 编写的所有教材和多媒体出版物:“Java How to Program”、“C How to Program”、“C++ How to Program”和“C & C++ Multimedia Cyber Classroom”。这个站点包括常见问题解答(FAQ)、错误分析、更新、附加的正文和范例,以及在程序设计和面向对象程序设计方面最新的技术进展情况。如果想更多地了解本书的作者或 Deitel & Associates 公司的有关情况,请访问 <http://www.deitel.com>。

## 1.2 什么是计算机

计算机是一种可以执行计算并可以进行逻辑判断的设备,它的运算速度高达每秒数百万、甚至数十亿次,远远高于人类的思维速度。例如,当今的个人计算机每秒可以完成数千万次的加法操作。一个人使用计算器需要数十年才能完成的数学运算,如果通过一台强大的个人计算机就可以在1秒钟之内完成。(注意:你怎么知道这个人计算正确?你怎么知道计算机算对了结果?)今天,最快的超级计算机可以在1秒钟完成几千亿次加法运算——大约相当于数十万人用一年时间进行的手工计算!而且每秒可执行万亿次指令的计算机已经在实验室中实现。

计算机通过所谓的计算机程序来处理数据。这些使得计算机按照一定的指令顺序执行的程序就是由程序员创造出来的。

那些构成计算机系统的不同设备(比如键盘、显示屏、磁盘、内存和处理单元)称为硬件。在计算机上运行的程序称为软件。这些年来,个人计算机已成为一种日用品,硬件的成本已大大地降低。遗憾的是,由于软件开发技术并没有什么进展,而且软件应用越来越复杂,功能越来越强大,因此软件的开发费用正在稳步上升。在本书中,读者将学会降低软件开发费用的方法——自顶向下逐步求精的方法、函数和面向对象的编程方法。

## 1.3 计算机的组织结构

忽略不同的物理外形,每一台计算机都可以分成以下六个逻辑单元或部分:

1. **输入单元。**这是计算机的“接收”部分。它从不同种类的输入设备中得到信息(数据和计算机程序),并把这些信息交给其他单元以便进行处理。今天,大多数信息是通过类似于打字机的键盘和鼠标输入的。在将来,也许大多数信息将通过语音或者电子扫描仪进行输入。
2. **输出单元。**这是计算机的“发送”部分。它将计算机处理过的信息放置在不同的输出设备上,使得这些信息在计算机之外可以得到。计算机的输出信息将显示在屏幕上、打印在纸上或者用来控制其他设备。
3. **内存单元。**这是一个快速存取、容量相对较小的计算机“数据仓库”。它将输入单元中获得的数据保留下来,一旦需要使用,则可以立即得到这些信息。内存单元还保存已经处理完但尚未由输出单元放置到输出设备的信息。内存单元通常称为内存或主存。
4. **算术和逻辑单元(ALU)。**这是计算机的“制造”部分。它用来执行诸如加、减、乘、除这样的运算。ALU包含判断逻辑,可以使计算机执行诸如比较内存中两个单元的内容是否相等这样的操作。
5. **中央处理单元(CPU)。**这是计算机的“管理”部分。它是计算机的调度员并且负责控制其

他单元。CPU告诉输入单元何时可以将信息读入内存,告诉ALU何时可以将信息从内存中取出来进行运算,告诉输出单元何时将信息从内存中取出并输出到合适的输出设备上。

6. **二级存储单元。**这是一个可以长期保存的、高容量的计算机“数据仓库”。程序或数据如果没有被其他单元使用,一般就放置在二级存储设备(例如磁盘)中。存放周期可能是几个小时、几天、几个月甚至几年。在二级存储设备中,信息保存的时间要比在主存中长得多。二级存储设备的单位费用比主存的单位费用低得多。

## 1.4 操作系统的发展

早期的计算机一次只能处理一件工作或任务。这种计算机操作过程通常称为单用户的批处理过程。计算机一次运行一个程序,处理一组或一批数据。在早期的系统中,用户通常使用穿孔卡片到计算中心去处理他们的任务。用户常常要等上几个小时甚至几天才能得到输出结果。

为了方便地使用计算机,称为操作系统的软件系统逐渐发展起来。早期的操作系统管理不同的批处理,使得它们之间可以顺利过渡。两个任务之间过渡的时间越短,计算机能够处理的工作总量(即生产率)就越大。

计算机的功能越来越强大,因此单用户的批处理方式约束了计算机资源的利用率。自然有人会想到,如果大量的任务共享机器资源,那么计算机的利用率就会高得多,这就是多道程序设计。多道程序设计同时对计算机中的许多任务进行操作——不同的任务共享并竞争计算机资源。在早期的采用多道程序设计的操作系统中,用户仍然通过穿孔卡片进行输入,为了得到结果也要等待几个小时甚至几天。

20世纪60年代,工业界和大学中的一些研究小组提出了分时操作系统的概念。分时可以认为是多道程序的一种特殊情况,用户通过终端(一种配有键盘和显示器的设备)来访问计算机。在一个典型的分时计算机系统中,数十个甚至数百个用户同时共享一台主机。计算机并不是同时运行所有用户的程序,而是运行某一用户程序的一小部分,然后再去执行另一用户的程序。这个过程非常迅速,系统可以在1秒钟之内为同一用户服务多次。这使得所有用户的程序好像都在同时运行。分时系统的优势在于用户随时都能够得到系统的响应,而不是像从前那样需要等待很长一段时间。

## 1.5 个人计算、分布式计算和客户/服务器计算

1977年,Apple公司将个人计算机普及开来。最初,拥有一台计算机只是爱好者的梦想,随着软件、硬件的发展,计算机变得如此经济实惠,以至于人们可以为了个人或商业上的用途而购买它们。1981年,世界上最大的计算机厂商IBM推出了IBM个人计算机。几乎就在一夜之间,个人计算机出现在各种商业、工业和政府机构中。

但是,这些计算机都是“独立”的个体,人们在自己的机器上完成自己的工作,信息的共享不得不靠磁盘复制而达到。虽然早期的个人计算机还没有能力采用分时的方式以提供给几个不同的用户使用,但它们有时可以通过电话线或者局域网而连在一起。这样,分布式计算机的概念逐步形成。在分布式计算环境下,一件事物的处理不再是集中在某一台中心计算机,而是通过网络将它分布到许多不同的设备上。个人计算机已经拥有足够的能力来处理单一用户的事务,同时它们还能够提供信息通信的基本手段。

今天,功能最强大的个人计算机的处理能力同10年前价值百万美元的设备相当。最强大的台

式机（称为工作站）为每个用户提供巨大的功能。信息在计算机网络上随意地共享，其中有一些机器称为文件服务器，它们为程序和数据提供巨大的存储空间；还有一些机器称为客户，它们通过网络使用文件服务器上的有关信息。这就是我们所说的客户/服务器结构。C 和 C++ 成为编写系统软件、网络软件和分布式软件应用的重要语言，而 Java 正成为开发基于 Internet 的应用的重要语言。下面的章节将讨论当今最流行的各种操作系统（包括 UNIX、OS/2、Windows 95 和 Windows NT）所能提供的各种功能。

## 1.6 机器语言、汇编语言和高级语言

程序员采用各种语言为计算机编写指令，有的语言可以直接在机器上执行，有的必须经过中间的翻译步骤。今天正在应用的语言有数百种，它们可以分为三种类型：

- (1) 机器语言
- (2) 汇编语言
- (3) 高级语言

每一台计算机只能直接理解它自己的机器语言。机器语言是每一台特定机器的“自然语言”，它由计算机硬件的设计者定义。机器语言通常由数字串组成（最终简化为 0 和 1），它将执行计算机中最基本的操作。机器语言是依赖于特定机器的，也就是一种机器语言只能用在某一种特定类型的计算机上。对于人类来说，机器语言过于繁琐。下面的例子说明了在某一种机器语言中，如何将 overtime（超时）的费用加在 base（基本）费用上，并将结果放入 gross（总和）费用中：

```
+1300042774
+1400593419
+1200274027
```

显然，随着计算机的普及，对于大多数程序员而言，机器语言编程不仅太慢而且枯燥乏味。为了代替那些计算机能够直接看懂的数字串，程序员们开始使用一种有些像英文缩写的助记符来表示计算机的基本操作。这些助记符构成了汇编语言的基础。称为汇编器的翻译程序可以用来在计算机上快速地将汇编程序转换为机器语言。下面的例子说明在汇编语言中如何将 overtime 的费用加在 base 费用上，并将结果放入 gross 费用中。不同的是，现在的指令看起来要比机器语言清楚得多：

```
LOAD BASEPAY
ADD OVERPAY
STORE GROSSPAY
```

尽管这些代码对于人们来说不难理解，但是在将它们经过汇编器翻译成机器语言之前，计算机是无法理解的。

随着汇编语言的发展，计算机应用迅速增长。但是，为了完成一个简单的任务，采用汇编语言仍然需要大量的指令。为了加快编程速度，人们开发了高级语言，这样使用一条语句就可以完成大量任务。将高级语言编写的程序转换成机器语言的翻译程序称为编译器。高级语言允许程序员采用接近日常英语的指令来编写程序，传统的数学符号同样可以使用。上述例子如果采用高级语言编写，则如下所示：

```
grossPay = basePay + overTimePay
```



从程序员的角度看来,高级语言显然要比汇编语言和机器语言的描述性更好。C、C++、Java就是使用最广泛、功能最强大的高级语言

将高级语言编写的程序编译为机器语言需要耗费计算机相当长的一段时间。于是一种称为解释器的程序用来直接执行高级语言编写的程序,而不必将它们编译成机器语言。尽管编译好的程序的执行速度远远快于在解释器中执行的程序,但在那些由于增加新的性能或者正在调试而需要频繁重复编译的开发环境中,解释器的应用仍然相当广泛。一旦程序开发完成,发行的将是一个运行更快的编译好的版本。

## 1.7 C++ 的历史

C++从C语言发展而来,而C语言又是从两种较早的语言BCPL和B发展而来。BCPL是由Martin Richards在1967年开发的,它是一种用来编写操作系统和编译器的语言。贝尔实验室的Ken Thompson对作为BCPL的一个副本的B语言进行了许多修改,然后于1970年在一台DEC PDP-7上用B语言编写出了UNIX操作系统的早期版本。BCPL和B语言都是“无类型”的语言——每一个数据项对应内存中的一个“字”。例如,程序员可以随意将一个数据项作为整数或是实数。

贝尔实验室的Dennis Ritchie于1972年在一台DEC PDP-11上实现了由B语言发展来的C语言。C语言在继承原有BCPL和B语言大部分特点的基础上增加了数据类型等其他特点。C语言最初是作为UNIX系统的开发语言而为人们广泛了解。今天,基本上所有的操作系统都是用C或C++编写的。在过去的20年中,大多数计算机上都实现了对C语言的支持。因此,我们可以说C是硬件独立的。只要谨慎地进行设计,就可以将C程序移植到大多数计算机上。

到了20世纪70年代末期,C语言发展成今天所谓的“传统C”或“Kernighan & Ritchie C”。1978年,Kernighan和Ritchie在Prentice Hall出版了“The C Programming Language”一书,本书引起了人们对C语言的广泛兴趣。这本书算得上是迄今为止最成功的计算机科学类书籍。

在不同类型的计算机(有时称为硬件平台)上广泛使用的C语言导致了许多变形。它们看起来都很相似,但往往互不兼容。程序员如果希望编写可以在几种不同平台上运行的且具有良好可移植性的C程序,那将会是一个很棘手的问题。显然,需要制定一个C语言标准。1983年,在美国国家标准协会(ANSI)所属的计算机与信息处理标准委员会(X3)指导下成立了XJ311技术委员会,并提供了一个无歧义性且设备独立的语言定义。这个标准在1989年再次发展。ANSI同国际标准化组织(ISO)进行了合作,在全球范围将C语言标准化;1990年出版了联合标准文档,称为ANSI/ISO 9899:1990。Kernighan和Ritchie参照ANSI C,于1988年出版了C语言一书的第二版,这本书描述的C语言已在全球广泛地使用(Ke88)。

作为C语言的扩展,C++是贝尔实验室的Bjarne Stroustrup于20世纪80年代初开发出来的。C++提供了大量完善C语言的特性,但更重要的是它提供了面向对象的编程功能。

人们总是过高地要求软件更新、更强,但是如何正确、迅速和高效地进行开发仍然令人难以捉摸。软件开发者认为,采用基于标准模块、面向对象的开发方法,比采用诸如结构化程序设计这样的传统开发方法更加高效。面向对象的程序更容易理解、调试和修改。

现在已经有其他许多种面向对象的语言,例如Xerox的Palo Alto研究中心(PARC)开发的Smalltalk语言。Smalltalk是一种纯粹的面向对象语言——理论上任何事物都是一个对象。C++是一种混合语言,可以用它来开发C风格的程序、面向对象风格的程序或者两者兼而有之。

## 1.8 Java 的历史

也许微处理器革命带给今天最大的成就是全球个人电脑将很快达到十几亿台。个人电脑已经对人们的生活和各种机构的运作方式产生了极大影响。许多人相信微处理器发挥巨大作用的下一个领域将是智能化的电器消费产品。正是出于这个认识，Sun Microsystems 公司（以下简称为 Sun 公司）于 1991 年开始了一个命名为“Green”的内部合作研究计划。这个计划最终决定开发一种基于 C 和 C++ 的语言，并称其为“Oak”（橡树），因为它的创造者 James Gosling 所在的公司窗外有一棵橡树。但是后来发现，已经有一种程序设计语言叫做“Oak”，于是 Sun 公司的员工在一次喝咖啡时想出了 Java 这个词，并且大家都接受了这个名字。

但是，“Green”计划遇到了麻烦。智能电器市场并没有按照人们想像的那样迅速地发展起来。更糟糕的是，Sun 公司丢掉了重要的合同，因此这个计划有被取消的危险。幸运的是，1993 年 WWW 迅速地发展起来。Sun 公司发现可以立即采用 Java 来创造含有动态内容的网络主页。这给“Green”计划带来了新生。

1995 年 5 月，Sun 公司在重要的会议上正式地宣布了 Java。起初，像这样的事情不会引起很大反响。但是，由于商业界对 WWW 产生了极大兴趣，因此 Java 也引起了他们的关注。Java 不同于 Pascal 这样的个人开发的语言，也不同于一个小组为了自己的使用目的而开发出来的 C 或 C++。Java 纯粹是为了商业目的而开发；并且随着 WWW 的迅速发展，Java 越来越引起人们的兴趣。

## 1.9 Java 的类库

Java 程序由类和方法构成。程序员可以自己编写出 Java 程序的每一部分；但是，大多数 Java 程序员利用了 Java 类库中的类和方法。在 Java 世界中两部分需要学习：第一个部分就是 Java 语言本身，可以用它来编写自己的类和方法；第二个部分是如何使用扩充的 Java 类库。在本书中，我们讨论了许多库中的类和方法。类库主要由编译器的供应商提供，但是许多是由独立的软件供应商提供的。

### 软件工程视点 1.1

采用构件块的方法建立一个系统，可以避免重复劳动。使用已有的部件编写程序（这称为软件重用）是面向对象程序设计的精髓。

### 软件工程视点 1.2

使用 Java 编程时，一般采用这样一些构建模块：类库中的类和方法，自己开发的类和方法，以及别人提供的、已经开发好的类和方法。采用自己的类和方法的好处是可以清楚地知道它们是如何工作的，可以测试每一行代码；但代价是为了设计新的类和方法将花费更多的时间和更大的精力。

### 性能提示 1.1

由于类库中的类和方法都经过了精心设计，以达到良好的运行效率，因此采用类库中的类和方法能提高程序的性能。

### 可移植性提示 1.1

由于几乎所有 Java 工具的类库中都含有同样的类和方法，因此采用类库中的类和方法会比用户自己编写的类和方法具有更好的可移植性。

### 软件工程视点 1.3

大量的可重用类库可以从 Internet（特别是 WWW）上获得。这些库的绝大多数都是免费的。

## 1.10 其他高级语言

尽管存在数百种高级语言,但是只有几种得到了广泛应用。FORTRAN (FORmula TRANslator) 是IBM公司于1954年~1957年开发出来的,它主要用于要求复杂数学计算的科学和工程应用。FORTRAN现在仍然被广泛地应用,尤其是在工程领域。

COBOL (COmmon Business Oriented Language) 是由一群计算机制造商以及政府和业界的计算机用户一起于1959年开发的。COBOL主要用于需要大量精确处理数据的商业领域中。今天,仍有超过一半的商业软件使用COBOL编写,有将近100万人仍在编写COBOL程序。

Pascal几乎与C语言同时开发出来,它是Nicklaus Wirth教授为了大学教学而开发的。我们将在下面的章节更多地介绍Pascal。

## 1.11 结构化编程

20世纪60年代,许多大规模的软件开发都遇到了严重的问题。软件开发计划常常延后,费用严重超支,并且最终产品无法使用。人们终于意识到软件开发是一件超出他们想像力的非常复杂的事情。经过20世纪60年代的研究,最终提出了结构化编程——采用这种较为严格的方法写出的程序同非结构化程序相比更加清晰,更易于调试和跟踪,并且更易于修改。在第3章中,我们将讨论结构化编程的主要原则。

在这项研究中,一个重要的收获是1971年Nicklaus Wirth开发出Pascal语言。该语言是以17世纪数学和物理学家Blaise Pascal的名字命名的,设计成用来在大学中教授结构化编程方法,并且迅速成为大多数大学愿意使用的编程语言。遗憾的是,由于它缺乏许多商业、工业和政府机构要求的功能特性,因此Pascal在这些领域并没有得到广泛应用。

Ada语言是在美国国防部(DOD)的资助下于20世纪70年代和80年代早期开发出来的。在这之前有数百种语言用来开发DOD规模庞大的命令和控制软件系统。DOD希望有一种单一的语言来满足它的大多数需要。Ada以Pascal为基本的参照,但是最终的Ada语言同Pascal截然不同。Ada语言是以诗人Lord Byron的女儿Ada Lovelace女士的名字命名的。Ada Lovelace于19世纪早期写出了世界上第一个计算机程序(用于Charles Babbage设计的分析机引擎的计算设备)。Ada语言的一个重要特性是多任务,这使得程序可以并发处理多项事务。而其他我们讨论过的广泛使用的高级语言(包括C和C++)都只允许程序在同一时刻执行一个任务。但是,Java语言支持多线程。

## 1.12 一个典型Java环境的基础知识

通常,一个Java系统包含以下几部分:一个环境,Java语言本身,Java应用程序接口(API),以及各种Java类库。图1.1详细解释了一个典型的Java程序开发环境。

Java程序一般分为五个阶段来执行(如图1.1所示),即编辑、编译、载入、验证和执行。如果读者使用的不是UNIX、Windows 95或者Window NT中的一种,虽然操作环境可能同图1.1所示的过程相类似,那么最好还是参考系统手册,或者询问教师如何在自己的操作环境中实现这些步骤。

第一个阶段包括编辑一个文件。这里应该使用一个编辑程序,程序员利用这个编辑器输入一个Java程序并进行必要的错误检查。然后这个程序存放在一个二级存储设备(例如磁盘)上。Java程序带有一个.java的文件后缀。UNIX系统常用的编辑器是vi和emacs。Windows 95和windows NT系

统提供的简单编辑器有DOS的Edit程序和Windows的Notepad程序,Java开发环境中集成了一个内置的编辑器。我们假设读者知道如何编辑一个程序

在第二个阶段,程序员采用javac命令来编译程序。Java编译程序将Java程序转换成字节码——Java解释器能够理解的语言。如果要编译一个名为Welcome.java的程序,只需键入:

```
javac Welcome.java
```

如果程序编译成功,将创建一个名为Welcome.class的文件。这个文件包含将在执行阶段解释的字节码。

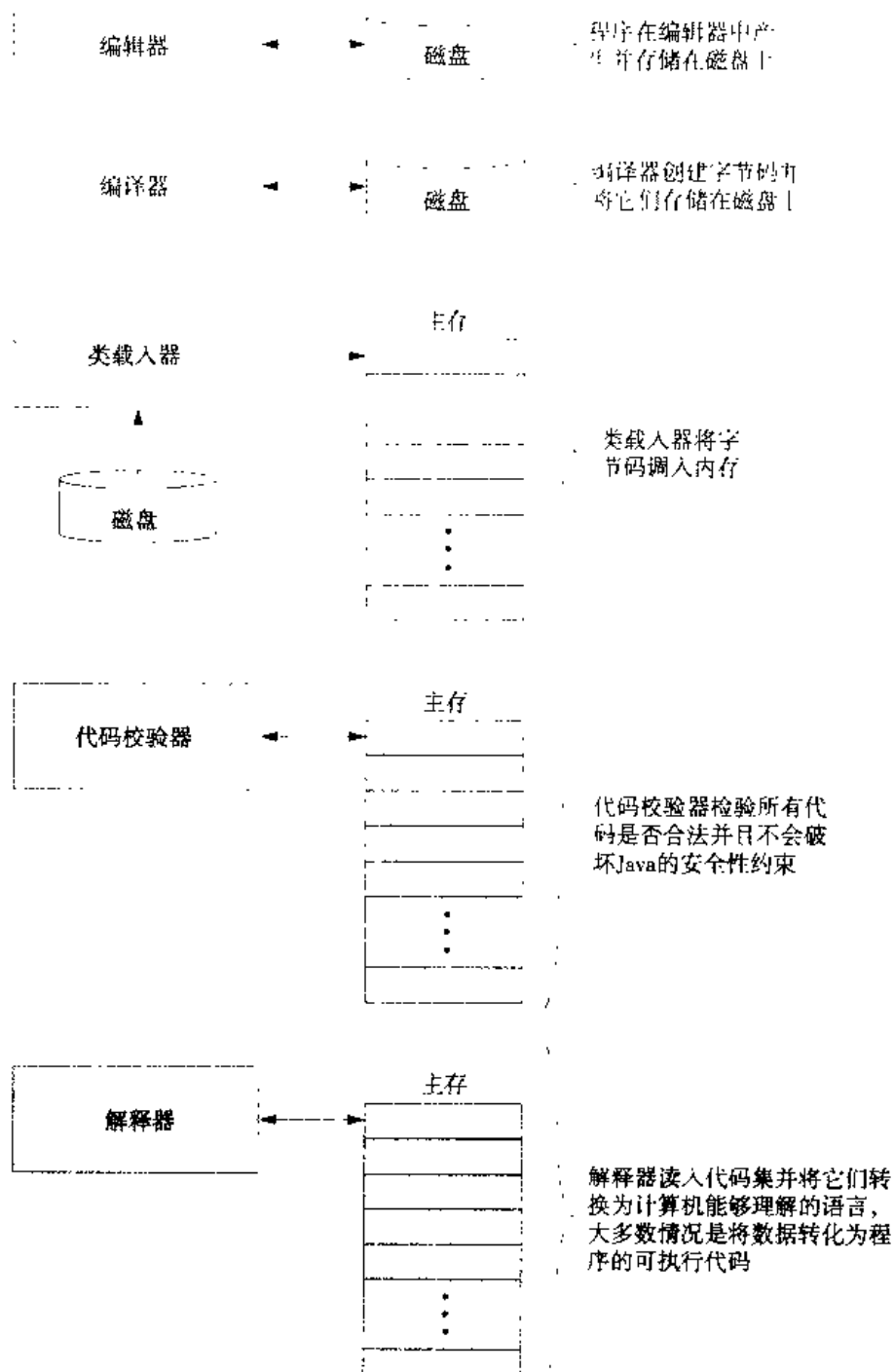


图 1.1 一个典型的 Java 环境

第三个阶段称为载入。程序在执行前必须首先放置到内存中。这个过程是由类载入器将一个或多个包含字节码的.class文件传输到内存来完成的。这些.class文件可以在本地硬盘上,也可以通过网络载入。在两种情况下类载入器都能处理.class文件。例如,命令“java Welcome”激活了java解释器,然后调用类载入器载入Welcome程序中使用的信息。Welcome程序称为一个应用程序,应用程序就是Java解释器执行的程序。当一个Java applet(小程序)由一个WWW浏览器(例如Netscape的Navigator或Sun的HotJava)载入时,同样也要激活类载入器。当用户浏览一个包含Java applet的HTML(超文本标识语言)文档时,将载入Java applet。Java applet还可以在命令行采用Java开发工具集提供的appletviewer命令来执行,这个工具集包括编译器(javac)、解释器(java)、appletviewer和其他Java程序员用到的工具。如同Netscape Navigator和HotJava,appletviewer需要一个HTML文档来激活一个Java applet。例如,如果Welcome.html文件包含Welcome这个Java applet,那么appletviewer命令将按以下方式使用:

```
appletviewer Welcome.html
```

这使得类载入器载入Welcome applet中用到的信息。通常我们将appletviewer看成最小的浏览器,它仅仅知道如何解释applet。

在Java解释器或appletviewer执行字节码之前,它们要在第四阶段由代码校验器进行验证。这样就保证了这些代码是合法的,并且不会破坏Java的安全性约束。由于Java程序在Internet上传播,为了避免破坏用户的文件和系统,必须保证严格的安全性约束。

最后是第五阶段,计算机在CPU的控制下逐字节地执行这个程序。

很少有程序在第一次执行后就能够正确工作。由于我们将要讨论的各种各样的错误,上述各个阶段都有可能失败。例如,一个正在执行的程序可能尝试去除以零(就像在数学中一样,在计算机中这也是一个非法的操作),这将导致计算机输出一个错误信息。程序员应该回到编辑阶段,修改相应的程序,然后继续执行,以判断修改是否有效。

#### 常见编程错误 1.1

像“除数为零”这样在程序运行时发生的错误,我们称之为运行时错误或者执行时错误。致命的运行时错误将导致程序在完成任务之前立即终止,非致命错误可能使程序继续运行,但常常导致不正确的结果。

大多数Java程序含有输入/输出。当我们说明一个程序输出一个结果时,通常是指将结果显示在屏幕上。数据可以输出到诸如磁盘和打印机这样的其他设备上。

## 1.13 预览本书

第1章“计算机和Java applet简介”。我们介绍了一些历史资料和一些简单的Java applet。本书使用了一种称为“活动代码”(Live Code)的技术,每一个概念都出现在这样的上下文中,即一个完整的可执行Java程序和它在计算机上执行时产生的输出结果。通过使用Java,我们可以写出两类程序,即“Java applet”用来在Internet上传播,并且可以在Netscape Navigator这样的WWW浏览器上执行;而单机的“Java应用程序”可用来在本地机器上运行。由于Java applet和图形用户界面更能引起大家的兴趣,因此在前几章中我们大多采用了Java applet。本书介绍了各种图形用户界面组件,这些组件将在第10章和第11章讨论。

第2章“控制结构(一)”。这一章我们关注的是程序的开发过程。本章讨论如何从一个问题陈述(即一份需求说明)开始,到最后开发出可执行的Java程序。本章介绍了一些基本的数据类型和

一些简单的用做条件判断的控制结构

**第3章“控制结构(二)”。**本章讨论Java中最像C的部分,包括顺序、选择和重复控制结构。本章使用一种简单的流程图来表述每一种控制结构,通过列举每一种控制结构而简要地介绍了结构化程序设计。第2章和第3章中讨论的技术覆盖了大学中所讲授的传统结构化程序设计的绝大多数内容。我们采用Java来寻求一种称为面向对象的编程方法。为了达到这个目的,我们会发现在实现对象的内部时,将会用到大量的结构化编程技术。

**第4章“方法”。**这一章我们将深入地研究对象内部。对象中包含的数据通常称之为实例变量,包含的可执行单元称为方法(在像C这样的非面向对象的过程语言中通常称为函数)。本章我们讨论了方法,包括方法调用本身,即递归调用的情况。

**第5章“数组”。**研究如何处理成列的数据和成表的数值的方法。数组在Java语言中将成为对象进行处理,这又一次证明了Java是100%的面向对象的语言。

**第6章“基于对象的编程”。**本章主要介绍了对象的要素和相关术语。什么是一个对象?什么是对象的类?一个对象的内部看起来像什么?对象是如何创建的?又是如何删除它们的?一个对象如何同其他对象通信?为什么一些打包软件中一个自然的结构类可以作为可重用的模块?

**第7章“面向对象的编程”。**本章讨论了类同对象之间的关系以及采用相关的类进行编程。我们如何才能找到类同对象之间的关系,以便降低编写大型软件的成本?什么是“一般的编程”而不是“特殊的编程”?为什么采用一般的编程方法写出的程序更易于修改或者更容易增加新的功能?我们怎样才能为对象的总体而不是为每一个不同类型的对象进行编程?

**第8章“字符串和字符”。**本章介绍如何处理字、句子、字符和字符数组。在这个问题上Java和C的关键区别是,Java将字符串作为对象处理,因此使用起来比C更加方便。而且更为重要的是,比起C使用危险的指针操作,Java的字符串操作则更加安全。

**第9章“图形”。**本章开始介绍Java中引人入胜的多媒体部分。传统的C或C++编程更加倾向于字符的输入/输出。一些版本的C++通过依赖于平台的类库来支持图形编程,但是采用了这些类库就表示该程序是不可移植的。Java图形功能的最精彩之处在于它是与平台无关的,可以移植。

**第10章“图形用户界面组件(一)”。**本章介绍如何创建具有用户友好的图形用户界面(GUI)的Java applet和应用程序。

**第11章“图形用户界面组件(二)”。**本章继续介绍了有关图形用户界面的内容。再一次强调,Java处理这些问题的方法是不依赖于平台的。一个基于GUI的Java applet或Java应用程序可以在所有不同的Java平台上运行。事实上,虽然不需要对Java的GUI应用程序做任何修改,但不同平台上的用户界面组件会表现出一些小小的差别,这也说明,Java紧密地同各种不同平台的GUI系统相关联,不论它们是Microsoft的Windows、Apple的Macintosh、Motif、Sun的OpenWindows、OS/2的Warp还是其他系统。所以,Java应用程序在各个不同平台上的表现将带给用户非常相似的感觉。

**第12章“异常处理”。**这是本书的重要内容之一,尤其是对于那些希望建立所谓关键任务和关键商业应用的用户。今天,当计算机的运行速度高达每秒1亿次的情况下,程序执行时出错是很常见的,而且错误来势极其迅速。程序员在使用构件时总是会问:“我怎样才能让一个构件为我实现某个功能?”他们还问:“这个构件返回什么值才表明它完成了我要求的工作?”但是程序员还需要关心这样一个问题:“如果我调用的构件在运行时遇到困难怎么办?如果它在运行时遇到问题怎样通知我?”在Java中,当一个构件(即一个类的对象)遇到了问题时,通常它能够抛出一个异常。有关那个构件的环境将设计成能够捕捉到这个异常并处理它。Java的异常处理功能特别符合当今的面向对象的情况,因为程序员构造的系统广泛地采用了其他程序员编写好的、可重用的构件块。在使用一个构件时,不仅要知道它正常运行时如何工作的,更要了解当它们出错时将抛出什么样的异常。

**第13章“多线程”。**这一章涉及到如何使Java applet和Java应用程序并行处理多件事务。尽管人们的身体能够同时进行很多工作(呼吸、吃饭、血液循环、观察等都是同时发生的),但是我们的思维却很难做到这一点。过去将计算机设计成带有一个昂贵的芯片,而今天的处理器却是如此便宜,以至于我们可以在一台计算机中同时使用多个芯片。这就是多处理器计算机,其发展趋势是计算机能够同时处理多个任务。今天的大多数程序设计语言,例如C和C++,并不支持并行操作。这些语言通常称为顺序程序设计语言或者单线程控制语言。但是Java拥有处理多线程的能力,即将应用程序设计成多个事务同时发生。这也说明Java在这个从20世纪90年代中后期兴起的基于多处理器、网络、多媒体的应用环境中具有更强的处理能力。

**第14章“多媒体:图像、动画和声音”。**本章关注Java在图像、动画、音频乃至视频方面的处理能力。值得注意的是,学生在一开始的课程中就要编写有关所有这些特性的程序。在Internet和各种CD-ROM上包含容量巨大的图像库,通过音频和视频我们可以将自己融入其中并创造丰富多彩的应用。如今,有超过一半的新出厂的计算机拥有多媒体功能。不久的将来,多媒体功能会同软盘驱动器一样普遍。

**第15章“文件和流”。**本章介绍通过数据流来实现到文件的输入/输出。对需要开发商业应用的程序员来说,这是最重要的语言特性之一,程序如何将数据传输到磁盘这样的二级存储设备上?程序如何从磁盘上读取数据?什么是顺序访问文件?什么是随机访问文件?什么是缓冲区?如何在大规模的读写操作中使用缓冲区来提高程序的运行效率?

**第16章“网络”。**本章介绍在计算机网络上通信的Java applet和Java应用程序。什么是客户?什么是服务器?客户如何呼叫服务器来完成它们的任务?服务器如何将结果传送给客户?什么是URL(统一资源定位器)?Java程序如何加载其他的WWW主页?我们如何采用Java来开发协作应用?

**第17章“数据结构”。**本章介绍如何将数据构造成为有用的集合,例如链表、堆栈、队列和树。每一种数据结构在广泛的应用领域都具有重要地位。我们将详细介绍每一种常用的数据结构。对于Java程序员来说,精心设计各种有用的类是很有价值的经验,同时也是非常重要的技巧。本章内容对于实现绝大多数的类都是非常有用的。尽管了解这些类如何工作是很重要的,但是Java程序员会很快发现,绝大多数的数据结构在现有的类库中都能提供,例如在下一章要讨论的Java本身带有的java.util软件包。第17章再一次强调了在第6章和第7章中讨论的、有关基于对象的编程和面向对象的编程中使用类的技巧。

**第18章“Java工具包和位处理”。**本章浏览了java.util软件包中的类。对于再一次强调“重用、重用、再重用”,本章进行了精辟的分析。如果可以重用已存在的类,那么将比重新开发程序更加快捷。这些类之所以包含在类库中,是因为它们是通用的、正确的、性能表现良好的、可移植的以及其他许多原因。许多人为了这些类已经付出了大量的劳动,为什么还要再做无用功呢?我们相信,世界上的类库将会以一定的速率不断增长。如果真是这样,那么作为一个程序员,你的技巧和价值将取决于你对现有的类的熟悉程度,以及是否知道如何重用它们来开发出高质量的软件。本章讨论了许多类,其中最有用的两个是Vector(一个可以增大或缩小的动态数组)和Stack(一种动态数据结构,仅仅允许在它的顶端操作,以保证后进先出的顺序)。这两种类是从第7章讨论的类中继承而来的,java.util软件包根据其他一些类来实现一些类,从而避免了重复开发并且利用了“重用、重用、再重用”的原则。

## 1.14 关于Java和本书的一般注意事项

Java是一种功能强大的语言,有经验的Java程序员有时以写出一些奇怪的程序为荣,这是一种

不好的编程习惯，只能造成程序难以阅读、表现奇怪、难以测试和调试，而且更难以适应更改的需求。本书同样面向初学者，所以我们特别强调清晰性。下面是我们给出的第一个“编程技巧”。

#### 编程技巧 1.1

采用简单明确的方式编写 Java 程序，有时我们称之为 KIS (Keep it simple, 保持简单)，不要刻意采用奇怪的用法。

我们将在本书中给出大量的编程经验，这些技巧可以帮助读者写出更加清晰、更好理解、更好维护、更易测试和调试的程序。这些经验仅仅作为参考，毫无疑问，读者可以按照自己的风格来开发程序。同时，我们还给出了**常见编程错误**（为了防止读者在今后的程序中犯类似的错误）、**性能提示**（帮助读者写出运行得更快、内存需求更少的程序）、**可移植性提示**（帮助读者写出只需很小修改甚至不用修改就可以在不同计算机平台上运行的程序）。这些技巧还包括如何全面观察 Java 拥有良好的可移植特性，并给出了**软件工程视点**（能够影响并提高一个软件系统、尤其是大型软件系统的思想和概念）。

Java 是一种可移植的语言，采用 Java 编写的程序可以在许多不同的平台上运行。可移植性是一个较难实现的目标。ANSI C 标准文档 (An90) 包含很大一部分关于可移植性的内容，并且整本书都用来讨论可移植的问题 (Ja89) (Ra90)。

#### 可移植性提示 1.2

尽管同其他大多数语言相比，采用 Java 更容易编写出可移植的程序，但是不同的编译器、解释器和计算机仍然可能对可移植性造成障碍。仅仅采用 Java 进行编程，并不能保证在所有情况下都是可移植的。程序员偶尔需要直接同编译器和不同种类的计算机进行交互。

我们仔细地浏览了 Sun 公司的 Java 文档，并且审查了它的完整性和正确性。尽管如此，由于 Java 是一种内容丰富的语言，仍然有一些细节和内容我们没有涉及。如果读者需要有关 Java 的技术细节，建议阅读 WWW 上最新的 Java 文档：<http://www.java.sun.com>。本书的参考文献中包含丰富的、讲述 Java 语言和面向对象编程的书籍和论文。

#### 编程技巧 1.2

阅读你正在使用的 Java 版本的说明文档，经常参考这个文档可以有助于读者了解 Java 的丰富特性以及是否正确使用了这些特性。

#### 编程技巧 1.3

计算机和编译器都是很好的老师。仅仅通过阅读 Java 的说明文档，并不能确保知道它是如何工作的以及产生了什么结果，仔细地研究编译 Java 程序以及调试 Java 程序时系统给出的信息，这有助于读者了解 Java。

在本书中，我们描述了 Java 在当今的应用平台中是如何工作的。也许早期的 Java 版本最明显的问题是，Java 程序是在客户端解释执行的。解释执行同编译好的机器码相比其速度要慢一些。但解释执行拥有一个极大的优势，那就是解释执行的程序可以下载到客户端并立即执行，而一个需要编译的源程序在执行前必须经过漫长的编译过程。目前在客户端仅提供 Java 解释器，但是大多数通用的计算机平台上都提供了编译器。这些编译器将 Java 的字节码编译成本机的机器码。这些编译好的 Java 程序与那些 C 和 C++ 程序相比，执行起来同样快捷。因此，不同的 Java applet 和应用程序之间就有所区别，所设计的在本地机执行的 Java 应用程序将是这些编译器的主要使用者。

Java applet 能够更多地吸引读者。请记住，Java applet 实际上可以从 WWW 中的任何一个服务器上获得。所以，Java applet 应该能够在任何可能的 Java 平台上运行。小型快速的 Java applet 仍



然通过解释执行。那些更加复杂、偏重于计算的Java applet 又该怎样运行呢？用户可能无法忍受等待的时间而要求更好的执行性能。对于一些特殊的偏重于性能要求的Java applet, 用户可能毫无选择余地，因为解释执行的Java 程序的速度太慢而达不到性能要求，所以Java applet 不得不通过编译执行。

一种介于解释执行和编译执行的方案是具有智能导向的解释器，也就是解释器通过执行代码来研究程序，对于那些需要加强执行的部分（例如循环）进行编译，并且执行这些部分的机器码而不再重新解释它们。

有关这些讨论的中心观点是，尽管现在Java 的字节码是通过解释执行的，但是利用正在发展的一些技术，我们可以使Java applet 达到更好的性能。策划者在评估Java 开发更重要信息系统的力量时，应该将这一点考虑进去。

对于想要开发大型信息系统的组织而言，主要的软件供应商已经开始提供集成的开发环境（IDE, Integrated Development Environment），集成开发环境提供了大量支持软件开发过程的工具。如今，市场上一些Java 集成开发环境同那些C 和C++ 的环境一样功能强大。这说明一些主要的IDE 供应商已经将Java 视为一种可以开发大量重要应用的、充满活力的语言。

Sun 公司已经公布了它的Java 数据库接口（JDBC）标准。这是专门为开发工业要求的数据库应用而准备的。它表明Sun 公司有意将Java 作为开发企业信息系统的手段。对于Java 来说，这是一个比开发活动网页更为重要的任务。

另一个有关Sun 公司支持Java 的例证是，Sun 公司公布了多种可以直接执行Java 字节码的微处理器的设计方案。这将大大增加直接解释执行Java 字节码的性能。最初的3 种Java 微处理器的价格从几美元到100 美元不等。Sun 公司所预计的这些微处理器的重要市场将包括最初设想的Java 市场——智能用户电子设备，例如电视的机顶盒、VCR、安全设备等。

## 1.15 Java 编程介绍

Java 语言可以用来帮助学习计算机程序设计。我们现在开始介绍Java 编程，并且通过几个例子来表现Java 的许多重要特性。每一个例子都会逐句分析。第2 章和第3 章将详细讲述Java 中的程序开发和程序控制。

## 1.16 一个简单的例子：打印一行文本

Java 使用的注释在非程序员看来可能有些古怪。我们从一个显示一行文本的简单Java applet 开始。一个Java applet 是一个程序，它可以在Netscape 的Navigator 或Sun 的HotJava 这样的WWW 浏览器上运行。这样的程序和它的输出如图1.2 所示。

这个程序表述了Java 语言的几种重要特性，我们将详细解释程序的每一行。为了阅读方便，程序中都加上了行号，但这些行号不是Java 程序的原有部分。第8 行执行了程序的“真正工作”，在屏幕上显示“Welcome to Java Programming!” 这一短句。现在，让我们按顺序考虑程序的每一行。第一行：

```
// A first program in Java
```

这一行以“//” 开始，表明本行是一个注释。程序员在程序中加入注释来使程序文档化以及增加程序的可读性。注释还可以帮助其他人阅读和理解你的程序。注释在程序执行时不会导致计算机执行

任何动作。Java 编译器将忽略注释，并且不会产生任何字节码。“A first program in Java” 注释简单地描述了程序的目的。一个以 “//” 开始的注释称为单行注释，因为该注释的作用范围在行尾结束。稍后我们将讨论其他两种注释方法，它们可以用来注释多行语句。注意，一个 “//” 注释可以在一行的中间开始，它能够使这一行的剩余部分（如第 2 行和第 3 行）成为注释。

```
1 //A first program in Java
2 import java.applet.Applet; //import Applet class
3 import java.awt.Graphics; //import Graphics class
4
5 public class Welcome extends Applet {
6     public void paint ( Graphics g )
7     {
8         g.drawString ( "Welcome to Java Programming !", 25 , 25 );
9     }
10 }
```



图 1.2 第一个 Java 程序

#### 编程技巧 1.4

每一个程序由一个表明本程序用途的注释开始。

Java 包括许多预先定义的类，并且在磁盘上将相关的类放入一个目录中，我们称之为软件包。这些软件包称为 Java 类库或者 Java 应用程序编程接口（API）。下面这两行：

```
import java.applet.Applet; //import Applet class
import java.awt.Graphics; //import Graphics class
```

是 import 语句，用来载入编译一个 Java 程序所需要的类。这些特定的行告诉编译器从 java.applet 软件包中载入 Applet 类，并且从 java.awt 软件包中载入 Graphics 类。当用户在 Java 中创建一个 applet 时，必须载入 Applet 类。载入 Graphics 类的目的是程序可以在屏幕上显示信息。

Java 的最强大的功能是 Java API 的各个软件包中含有数目庞大的类，程序员可以重用它们而不必重新开发。我们在本书中运用了这些类的大量成员，并且将详细讨论类和软件包。

下面的语句：

```
public class Welcome extends Applet{
```

开始将 Welcome 类定义为一个 Java Applet 类。关键字 class 在 Java 中引入一个类的定义，紧接在后面的的是类的名称（本程序中为 Welcome）。每一个 Java 程序至少包含一个类的定义。当创建一个类的定义时，常常需要使用一个已存在的类定义的片段。由于这个原因，读者恐怕不能完全重新设计一个类。Java 提供了一种所谓的继承机理，使得用户在创建新的类时使用已有的类定义。关键字 extends 紧跟在类的名称后面，指出我们的新类是从哪里继承而来。在这个例子中，采用继承的方法使得我们的 Welcome 类既有新加入的特性，同时又具有它继承的 Java API 的 Applet 类中已经定义好

的各种功能。继承的机理很容易使用，程序员不需要非常了解被继承的类的许多情况。所有的Java applet必须从Applet类中继承而来，但是程序员不需要完全了解如何使用Applet类（本书将从头到尾地讨论Applet类）。在继承关系中，Applet称为超类或基类，Welcome称为子类或派生类。我们将在第7章“面向对象的编程”中详细讨论继承关系。

#### 编程技巧 1.5

从类中继承一个Java API的子类之前，请仔细阅读它的说明文档以了解这些类的功能。

类用来在程序执行时在内存中实例化（或创建）一个对象。一个对象是内存中的一个区域，其中存储程序使用的信息。我们的Welcome类用来创建一个执行这个Java applet的对象，其中运行这个applet的浏览器创建了Welcome类的一个实例。关键字public使得浏览器能够创建并执行我们的applet。在Java中，主要的Applet类（即从Applet中继承的）总是一个public（公有）类型的类。关键字public将在第6章“基于对象的编程”中详细讨论。

当把applet存入文件时，这个applet的类名将作为文件名的一部分。在本例中，文件名为Welcome.java。

#### 常见编程错误 1.2

如果文件名在拼写或缩写上同类名都不吻合，则将产生语法错误。

#### 常见编程错误 1.3

如果一个文件包含一个applet的类定义，那么它的文件名后缀必须为.java，否则将出现错误。

左花括号（{）表明一个类定义的开始。对应的右花括号（}）表明类定义的结束。下列语句：

```
public void paint( Graphics g )
```

引入了一个称为方法的程序构造块，我们将在第4章详细解释方法。Java程序至少包含一个方法，用来在applet执行时履行某个任务。图1.2所示的程序包含了paint方法（第6行~第9行）的定义。当一个applet执行时，将自动调用paint方法，从而在屏幕上显示信息。方法能够执行某个任务并在完成自己的工作后返回信息。关键字void表明这个方法将执行某一任务，但是当它结束时并不返回值。paint之后的左括号定义了这个方法的参数表，方法从此参数表中接收完成任务所需的信息。方法paint的参数表说明它需要一个Graphic的对象（名为g）来完成相应的任务。关键字public是必须的，这样浏览器可以自动调用定义的paint方法。到目前为止，所有的方法定义都是以关键字public开头的。

第7行的左花括号表明这个方法定义的程序体的开始。一个相应的右花括号表明这个方法定义的程序体的结束（程序的第9行）。

下列语句：

```
g.drawString("Welcome to Java Programming!",25,25);
```

指示计算机执行一个动作，即采用Graphics对象g的drawString方法画出引号中的字符串。一个串有时称为一个字符串、一条信息或一行文字。这样的一整行，包括g.drawString、括号中的参数以及分号（;），通常称为一条语句。每条语句必须以一个分号结束（称为语句结束符）。当执行上述语句时，它将在屏幕坐标的(25,25)点开始显示信息“Welcome to Java Programming!”。坐标是从屏幕上这个applet的左上角按像素计算的。一个像素（图形元素）是在屏幕上显示输出时的基本单位。在彩色显示器上，一个像素代表屏幕的一点。例如，许多个人计算机在屏幕的宽度方向上有640个像

素, 高度方向上有 480 个像素, 那么总数为  $640 \times 480$ , 即 307 200 个可显示的图形元素。许多其他种类的计算机则有更高的屏幕分辨率。屏幕的分辨率越高, Java applet 在屏幕上看起来就越小。图形坐标的第一个坐标是  $x$  坐标, 代表屏幕上 applet 从左向右数的像素个数; 第二个坐标是  $y$  坐标, 代表从上向下数的像素个数。大多数 Graphics 类中的画图方法要求至少有一个坐标集作为参数, 告诉程序在 Java applet 上的什么位置开始画图。

#### 常见编程错误 1.4

如果在行尾忽略了分号, 则将产生语法错误。

当编译器无法解析一条语句时将引发一个语法错误。编译器通常发布一个错误信息, 以帮助程序员定位和改正不正确的语句。语法错误是由于对语言的不正确使用而导致的。语法错误也称为编译错误、编译时错误或者编辑错误, 因为它们都是在编译阶段产生的。

#### 编程技巧 1.6

在每一个逗号(,)后面增加一个空格可以增加程序的可读性。

#### 编程技巧 1.7

在开始每一级结构时, 采用缩进格式来表明定义一个方法的结构体。这样可以清晰地突出程序的结构并且有助于阅读程序。

#### 编程技巧 1.8

最好将你喜欢的缩进格数固定下来, 并且坚持这一标准。可以使用 tab 键来缩进, 但是 tab 键的跳格数目可能会有所变化。我们推荐使用 1/4 英寸的 tab 键跳格数或者三个空格的缩进长度。

编译完图 1.2 所示的程序后, 必须创建一个 HTML (超文本标识语言) 文档来载入这个 applet 程序, 以便浏览器可以执行它, 浏览器将阅读文本文件。图 1.3 包含了一个简单的 HTML 文件 Welcome.html, 用来载入图 1.2 定义的 applet。

许多 HTML 代码标记 (tag) 是成对出现的。例如, 图 1.3 中的第 1 行和第 4 行指出文件中 HTML 代码的起止位置。所有的 HTML 标记由一个左尖括号 (<) 开始, 并由一个右尖括号 (>) 结束。第 2 行和第 3 行是专为 Java applet 使用的 HTML 标记。它们告诉浏览器载入某一特定的 applet 并规定它的大小。这个 applet 标记由几个部分组成。第一部分 (code = "Welcome.class") 表明哪一个文件包含定义的已编译的 Java applet。当编译 Java 程序时, 每一个类将编译成一个单独的文件, 并以类名和 .class 后缀作为文件名。Java applet 标记的第二部分和第三部分表明了这个 applet 执行时按像素计算的宽和高。Java applet 的左上角通常定义为  $x$  坐标和  $y$  坐标的原点。这个 applet 的宽为 275 个像素, 高为 35 个像素。第 3 行的 </applet> 标记同第 2 行的 <applet> 标记相对应, 表示这一段结束。第 4 行的 </html> 标记同第 1 行的 <html> 标记相对应, 表明全段结束。

为了示范我们的 applet, 我们使用了 Sun 公司的 Java 开发工具箱 (JDK) 中的 appletviewer 命令。appletviewer 是一个测试 Java applet 的程序。appletviewer 仅能识别 HTML 中的 applet 标记, 所以有时也称之为“最小的浏览器”(忽略了所有其他 HTML 标记)。appletviewer 是按以下方式从机器的命令行激活 Welcome Applet 类:

```
appletviewer Welcome.html
```

注意, appletviewer 需要一个 HTML 文件来载入 Java applet。

```
1 <html>
2 <applet code = "Welcome.class" width = 275 height = 35>
```

```
3 </applet>
4 </html>
```

图 1.3 Welcome.html 文件将图 1.2 中的 Welcome Applet 类载入一个浏览器中

#### 常见编程错误 1.5

如果 appletviewer 命令之后所跟的文件名后缀不是.html, 就不会执行 appletviewer。

可以使用好几种方法来显示“Welcome to Java Programming!”。在 paint 方法中使用两个 drawString 语句可以产生多行显示结果, 如图 1.4 所示。

```
1 //Displaying multiple strings
2 import java.applet.Applet;    //import Applet class
3 import java.awt.Graphics;      //import Graphics class
4
5 public class Welcome extends Applet {
6     public void paint ( Graphics g )
7     {
8         g.drawString("Welcome to ", 25, 25);
9         g.drawString("Java Programming", 25, 40);
10    }
11 }
```

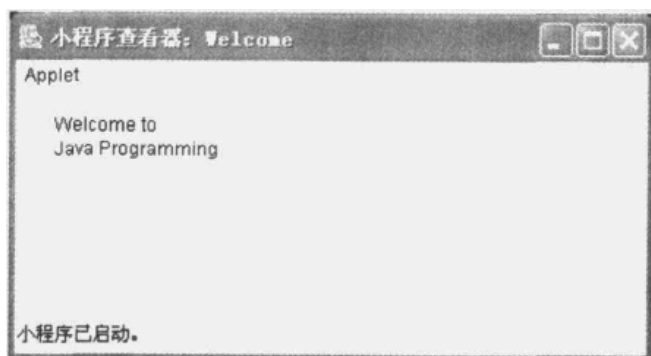


图 1.4 显示多行字符串

## 1.17 另一个 Java 程序：整数相加

下一个程序是由用户在键盘上输入整数, 然后计算机计算出结果并将它显示出来。当用户键入每一个整数并按下回车键时, 将由程序读入这个整数并加入到总数中。

这个程序使用了 Java API 中一些图形用户界面 (GUI) 的组件。GUI 组件可以帮助程序读入数据并输出数据。例如, 图 1.5 包括一个 Netscape Navigator 的窗口。在此窗口中, 有一栏称为菜单栏 (File、Edit、View 等) 的自选项。菜单栏之下有一些按钮, 它们分别对应 Netscape Navigator 中的某个任务。在这些按钮之下有一个文本字段, 用户可以在这里输入他们想要访问的 WWW 站点地址。文本字段的左方是一个标记, 它表明这个文本字段的用途。菜单、按钮、文本字段和标记都是 Netscape Navigator 的 GUI 的组成部分, 它们使用户能够交互使用 Navigator 程序。Java 所包含的实现 GUI 的组件将在本章以及第 10 章、第 11 章中讨论。我们的下一个 Java applet (如图 1.6 所示) 使用了一个标记和一个文本字段, 保证用户可以交互使用这个 Java applet。

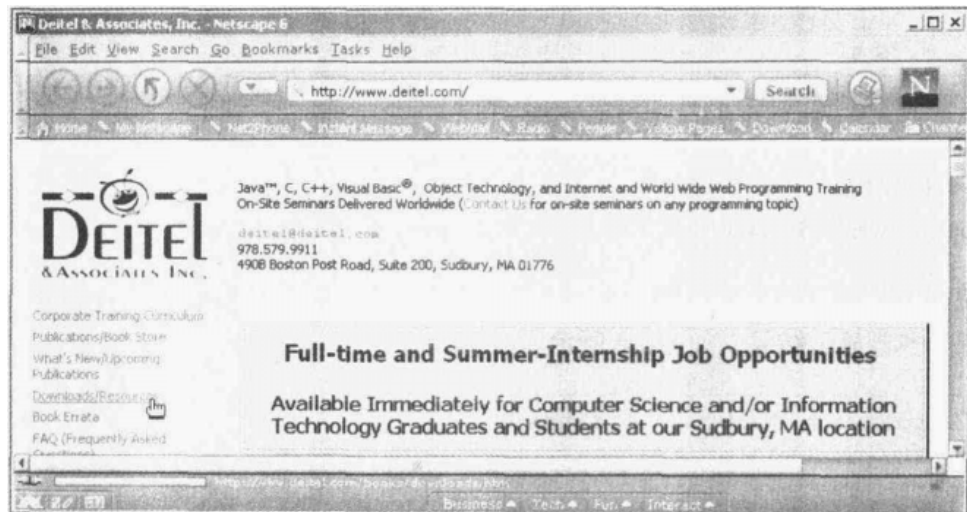


图 1.5 一个拥有 GUI 组件的简单的 Netscape Navigator 窗口

```

1  //Addition program
2  import java.awt.*;           //import the java.awt package
3  import java.applet.Applet;
4
5  public class Addition extends Applet {
6      Label prompt;           //prompt user to input
7      TextField input;        //input values here
8      int number;             //store input value
9      int sum;                //store sum of integers
10
11      //setup the graphical user interface components
12      //and initialize variables
13      public void init()
14      {
15          prompt = new Label( "Enter integer and press Enter:" );
16          input = new TextField( 10 );
17          add( prompt );       //put prompt on applet
18          add( input );        //put input on applet
19          sum = 0;             //set sum to 0
20      }
21
22      //process user's action on the input text field
23      public boolean action( Event e, Object o)
24      {
25          number = Integer.parseInt( o.toString() ); //get number
26          input.setText( "" ); //clear data entry field
27          sum = sum + number; //add number to sum
28          showStatus( Integer.toString( sum ) ); //show result
29          return true;         //indicates user's action was processed
30      }
31  }

```



图 1.6 一个正在执行的加法程序

下面的注释:

```
//Addition program
```

说明了图 1.6 所示程序的用途。

下面这一行:

```
import java.awt.*; //import the java.awt package
```

载入 java.awt 软件包, 这样编译器才可以编译这个 applet。星号 (\*) 表明 java.awt 软件包中所有的类在编译时都将提供给编译器。记住, 我们前两个程序仅仅载入了 java.awt 软件包中的 Graphics 类。在这个程序中, 使用了 java.awt 软件包中的 TextField 和 Label 两个类。当把整个软件包载入程序时, 编译器仅仅载入那些程序中用到的类。

如前所述, 每一个 Java 程序都至少拥有下面这样的类定义, 类定义是通过继承的方法建立在原来已有的某一个类定义的基础之上。记住, Java applet 必须从 Applet 类继承。下面的语句:

```
public class Addition extends Applet {
```

表示 Addition 类从 Applet 类继承而来。所有的类定义由一个左花括号开始, 以一个右花括号结束。

接下来的语句:

```
Label prompt;           // prompt user to input
TextField input;        //input values here
int number;             //store input value
int sum;                //store sum of integers
```

为声明语句。prompt 和 input 是两个引用, 指向程序中的 Label 和 TextField 对象。一个对象包含程序所需的信息。我们在将来学习基于对象和面向对象的编程时, 将会更详细地定义对象的概念。number 和 sum 是变量的名称。一个变量同对象是相似的, 两者的主要区别在于一个对象由一个类所定义, 其中可以同时包含信息和方法; 而一个变量由一个原始的 (或称为内置的) 数据类型所定义, 例如 int 类型 (代表整数值, 如 7、-11、0、31 914 这样的整数)。Java 程序中的每一个数据都可以

看做一个对象，但是由原始数据类型定义的变量除外。我们将在第二章讨论其他的原始数据类型。

每一条声明语句以一个分号结束，可以声明一个或多个引用或变量。input 引用的类型是 TextField。文本字段用来从键盘接收用户的输入，并把信息显示在屏幕上。prompt 引用的类型是 Label。一个 Label 包含一个将显示在屏幕上的字符串。通常，一个标签同屏幕上的另一种图形用户界面组件相联系（例如在本程序中是文本字段）。变量 number 和 sum 的类型是原始数据类型 int。所有的变量和引用都必须在程序使用之前声明一个名称及其类型。同一类型的变量或引用可以在一条语句行中声明，也可在多个语句行中声明。如果一个声明语句中包含不止一个变量或引用名称，那么名字之间要用逗号（,）分开，我们称之为逗号分隔表。

一个变量或引用的名称可以是任何有效的标识符。一个标识符是这样一串字符：包含字母、数字、下划线（\_）以及美元符号（\$），并且不以数字开头。Java 允许任何长度的标识符。Java 是大小写敏感的，即区分大写字母和小写字母，因此 a1 和 A1 是不同的标识符。

#### 编程技巧 1.9

选择有意义的变量名有助于一个程序具有“自我文档化”的功能，即不用阅读说明文档或使用额外的注释，仅仅通过阅读源代码就可以理解整个程序。

第 6 行～第 9 行声明了可以在整个类定义中（即这个类的所有方法）使用的名称。同样，可以在一个方法的定义中放置声明。但无论如何都应该在方法使用之前定义变量，在方法中定义的变量无法由其他方法直接使用，我们将在第 4 章深入讨论有关方法的主要内容。

#### 编程技巧 1.10

通常在声明和可执行语句之间用一个空行隔开，这样可以使声明部分更加醒目并且增加程序的清晰性。

#### 编程技巧 1.11

如果想在方法开始部分放置一个声明段，请使用一个空行将它同可执行部分隔开，这样可以突出声明部分的结束位置和可执行部分的开始。

这个 applet 包含两个方法——init（从第 13 行开始定义）和 action（从第 23 行开始定义）。方法 init 通常用来在一个 applet 中初始化这个 applet 用到的变量和引用。init 方法在一个 applet 开始执行时将自动调用，并且它是首先调用的方法。实际上，applet 常常按顺序调用三个方法：init、start（后面的例子将要讨论）和 paint。如果没有为这三个方法中的某一个（或多个）进行定义，则将从 Applet 类中得到一个“免费”的版本，但是它们不做任何操作，这也是我们所有的类都从 Applet 类继承而来的一个原因。action 方法在本程序中处理用户和图形用户界面之间的交互。

init 方法的第一行通常如下所示：

```
public void init()
```

这表明 init 是一个 public 的方法，不接收任何参数，并且在执行完成后也不返回任何值。

下面这行：

```
prompt = new Label("Enter integer and press Enter:");
```

创建一个 Label 对象，采用“Enter integer and press Enter:”这个字符串进行初始化。该字符串将作为 applet 中的 input 文本字段的标签。这条信息称为一个“提示”，因为它告诉用户去完成一个特定的操作。运算符 new 在程序运行时创建一个对象，它需要向计算机申请足够的内存来放置 new 右边所指定的类型对象。在类型名后的括号中的值用来初始化（赋一个值）这个创建的对象。引用 prompt



通过赋值运算符(=)得到new操作处理后的结果。这条语句可以读为“prompt得到new Label(“Enter integer and press Enter:”)的值”。赋值运算符称为二元运算符,因为它含有两个操作数——prompt和表达式“new Label(“Enter integer and press Enter:”)”的值。

下面这行:

```
input = new TextField( 10 );
```

使用运算符new创建了一个TextField对象,并将结果赋给input。数字10设定了屏幕上此文本字段按字符计数的长度。

下面这两行:

```
add( prompt );  
add( input );
```

将Label和TextField这两个组件放在applet中,这样用户可以交互使用它们。add方法是我们从Applet类中继承而来的众多方法中的一个。而“sum = 0;”是一个赋值语句,它将sum的值置为0。注意,变量number和sum没有使用运算符new来生成,这说明只有类的对象必须用动态方式(new)申请空间,原始类型的变量由Java自动分配空间。

下面我们说明方法action——这是本书将要讨论的几个方法之一,它提供用户同applet的GUI组件之间的交互操作。所有在一个用户和一个GUI组件之间的交互操作都将产生一个事件。事件告诉Java applet,用户正在同它的某一GUI组件之间发生交互操作。例如,当用户从键盘上向文本字段输入一个数据并且按下回车键时,一个事件(实际上它是java.awt软件包中Event类的一个对象)将发送给applet,指出用户在文本字段中输入了回车键。这样将自动激活并执行action方法。在这个程序中,仅当用户在文本字段按下回车键时才调用action方法。这种用户同GUI组件之间的交互操作过程常常称为事件驱动的编程。

action方法的第1行:

```
public boolean action ( Event , Object o )
```

指出action是一个public的方法,它在执行完成后返回一个boolean(布尔类型,只有true和false两个值的一种原始数据类型)值。在本书前面的例子中,action方法总是返回一个true值。action方法有两个参数——Event和Object。参数Event用来确定发生了什么事情。我们将在下一个例子中讲解参数Event。

参数Object包含同事件相关联的信息。例如,当用户在文本字段按下回车键后,参数Object将包含刚才文本字段中的内容。参数Object可以是任何类型(原始数据类型除外)的对象,它是Java语言中最通用的类型(我们将在第6章和第7章中详细讨论Object)。Java的一个很好的特性是,任何类型的对象都可以采用toString方法而转化为String类型。String是一个可以存储字符串(第8章将详细讨论字符串)的对象。

一旦进入action方法,我们必须处理用户在文本字段中键入的信息。在action方法的程序体中:

```
number = Integer.parseInt( o.toString() );//get number
```

这一行将Integer.parseInt方法的结果赋给number, Integer.parseInt的作用是将其String类型的参数转化为一个整数。该方法所转化的这个字符串,是通过o.toString()方法将这个Object类型的对象转化为String类型而得到的。任何在程序后面使用number变量的位置都将用到同一整数值。

下面这一行:

```
input.setText( "" );//clear data entry field
```

采用 `TextField` 类中的方法 `setText`, 将文本字段 `input` 的内容置为空串, 即这个字符串中将不包含任何字符。这将清空用户在文本字段中输入的内容。

赋值语句:

```
sum = sum + number ;//add number to sum
```

计算变量 `sum` 和 `number` 的和, 并且使用运算符 “=” 将结果赋给变量 `sum`。这个语句可以读为 “`sum` 得到 `sum + number` 的值”。大多数计算将在赋值语句中出现。

计算完毕后, 我们在 `appletviewer` (或读者的浏览器) 窗口下方的状态栏上显示结果, 在图 1.6 的 4 个输出中, `appletviewer` 的状态栏在窗口的下方分别显示 “Applet started” (中文界面为 “小程序已启动”)、 “45”、 “72”、 “117”。下一行:

```
showStatus( Integer.toString( sum ) );//show result
```

使用 `Applet` 类的方法 `showStatus` 在状态栏上显示一个 `String` 类型。显示的 `String` 类型是使用方法 `Integer.toString(sum)` 将整数 `sum` 转化为 `String` 类型而得到的。

最后一行:

```
return true;//indicates user's action was processed
```

告诉 `action` 在执行完毕后返回 `boolean` 类型的 `true` 值。这也就告诉程序, 用户对该事件的处理已经完成。正如前面提及的, `action` 方法在我们早期的例子中将返回 `true` 值。

#### 编程技巧 1.12

在每个二元运算符的两侧各放置一个空格, 这样可以突出运算符并使程序易读。

## 1.18 关于内存的概念

`sum` 和 `number` 这样的变量实际上是对应于计算机内存的某一位置。每一变量都有一个名字、一个类型、一个存储长度和一个值。

在图 1.6 的加法程序中, 当执行语句 “`number = Integer.parseInt( o.toString() );`” 时, 用户在文本字段中键入的值将放置在编译器给变量 `number` 分配的内存空间中。假设用户键入数字 45 作为 `number` 的值, 计算机将如图 1.7 所示的那样把 45 放入内存空间中。

无论一个内存空间是否有值, 新的值将取代其中原有的值, 原有的值将消失。

一旦程序获得一个值, 它便将此值同 `sum` 相加, 并将结果放在变量 `sum` 中。下列语句:

```
sum = sum + number;
```

在执行加法的同时也覆盖了变量原有的值, 这将发生在 `sum` 和 `number` 的和已经计算完成、新的值置入了 `sum` 的内存地址的时候 (不论原来 `sum` 中存放了什么值)。`sum` 的计算完成后, 内存的状态如图 1.8 所示。注意, 变量 `number` 的值与执行加法之前是相同的。这个值由程序使用, 但是在计算机完成加法运算之后并没有改变原有内容。也就是说, 当我们从一个内存空间读出一个值时, 并不破坏原来的内容。



图 1.7 显示变量名字和值的内存空间

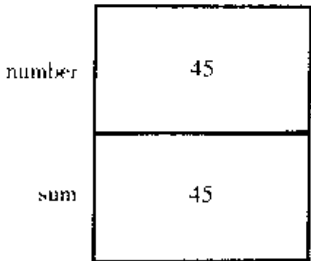


图 1.8 计算完成后的内存空间

1.19 算术运算

大多数程序都要进行算术运算。算术运算符的总表如图 1.9 所示，注意有一些特殊的符号在传统的代数符号中是没有的。星号（\*）表示乘，而百分符号（%）表示取模操作，下面我们将讨论这个运算符。图 1.9 中的算术运算符都是二元运算符，例如，表达式 “sum + value” 包括二元运算符 “+” 和两个操作数 sum 和 value。

Java 操作	算术运算符	代数表达式	Java 表达式
加	+	f+7	f+7
减	-	p-c	p-c
乘	*	bm	b*m
除	/	$x/y$ 或 $\frac{x}{y}$ 或 $x \div y$	x/y
取模	%	r mod s	r%s

图 1.9 算术运算符

整除表示按整数进行除法，例如，表达式 7/4 的值为 1，而 17/5 的值为 3。注意，任何除不尽的部分都将简单地忽略（即删除），而不进行四舍五入。Java 提供了取模运算符 “%”，它可以得到整除之后略去的余数。取模运算符是一个整数运算符，它仅仅用在整数之间。表达式 x%y 得到 x 除以 y 之后的余数。因此，7%4 等于 3，而 17%5 等于 2。在后面的章节中，我们考虑取模运算符的许多有趣用法，例如它可以用来判断一个数是否为另一个数的倍数。

常见编程错误 1.6

在取模运算符的两端出现非整数将导致语法错误。

算术表达式在 Java 中必须写成一 行，这样程序才能理解。因此，像 “a 除以 b” 这样的表达式必须写成 a/b 的形式，并且所有的常量、变量和运算符都在一行上。下面这种代数表示法：

$$\frac{a}{b}$$

通常编译器是不会接受的，尽管一些特殊用途的软件包为了进行复杂的数学计算而支持传统的代数

表示方法。

在Java表达式中,括号的用法同传统代数中的用法完全一致。例如,为了用a乘以b+c的和,我们可以写为:

`a*(b+c)`

Java在计算算术表达式时按照以下的运算符优先级顺序,这个顺序同传统代数中的约定相同。

1. 括号中的表达式首先得到计算。因此,括号可以由程序员用来实现特定的任何顺序的计算。括号拥有“最高优先级”。在含有嵌套或存在层次关系的前提下,内层括号中的表达式先得到计算。
2. 接下来是乘、除和取模运算符。如果一个表达式包含几个这样的运算符,那么按照从左到右的顺序进行计算。乘、除和取模运算符的优先级是相同的。
3. 加、减法运算符的优先级最低。如果一个表达式包含几个加、减法运算符,同样按照从左到右的规则进行计算。加、减法运算符拥有同样的优先级。

这些算术优先级规则使得Java正确地进行数学计算。当我们陈述从左到右运用运算符时,指的是同一优先级的不同运算符。我们也将看到,一些运算符是从右到左结合的。图1.10总结了这些运算符的优先级顺序。当我们接触到更多的Java运算符时,这个表会不断扩大,本书在附录A中给出了一个完整的优先级表。

运算符	操作	优先级
()	括号	首先计算。如果括号是嵌套的,内层括号中的表达式先得到计算。如果同一层次上(即非嵌套的)有好几对括号,则按从左到右的规则计算
*, / 或 %	乘 除 取模	第二优先级。如果多个运算符同时出现,则按照从左到右的规则计算
+ 或 -	加减	最低优先级。如果多个运算符同时出现,则按照从左到右的规则计算

图 1.10 算术运算符的优先级顺序

现在,让我们按照运算符的优先级顺序来查看几个表达式。每一个例子都有代数表达式和Java表达式相对应。

下面是一个取5个数的平均值的例子:

代数表达式:  $m = \frac{a+b+c+d+e}{5}$

Java表达式: `m = (a + b + c + d + e) / 5;`

这里要求有括号,因为除法的优先级高于加法。加完的总和(a+b+c+d+e)除以5。如果错误地忽略括号,我们将得到a+b+c+d+e/5的值,这相当于:

$a+b+c+d+\frac{e}{5}$

下面是占一行的等式的例子:

代数表达式:  $y = mx + b$

Java表达式: `y = m*x + b;`

不需要任何括号，上式将首先计算乘法，因为乘号的优先级高于加号。

下面的例子包括取模、乘、除、加和减法运算符：

代数表达式： $z = pr \% q + w/x - y$

Java 表达式： $z = p * r \% q + w / x - y;$



圆圈中的数字表示 Java 在实际计算时的执行顺序。乘、取模和除运算符首先按照从左到右的顺序（即它们按照从左到右的顺序结合）进行计算，因为它们的优先级高于加和减运算符。然后进行加法和减法运算，它们同样遵照从左到右的顺序。

并非所有表达式的多对括号都是嵌套的。例如，下列表达式：

$a * (b + c) + c * (d + e)$

并不包含嵌套的括号，这里的括号在“同一层次上”。

为了更好地理解优先级顺序的规则，我们看看下面这个二次多项式是如何计算的：

$y = a * x * x + b * x + c;$



圆圈中的数字表明 Java 实现时的操作顺序。由于在 Java 中不存在求幂运算符，我们将  $x^2$  写成  $x*x$ 。

假设变量 a、b、c 和 x 的初值为：a=2，b=3，c=7，x=5。图 1.11 说明了计算这个二次多项式的运算符优先级顺序。

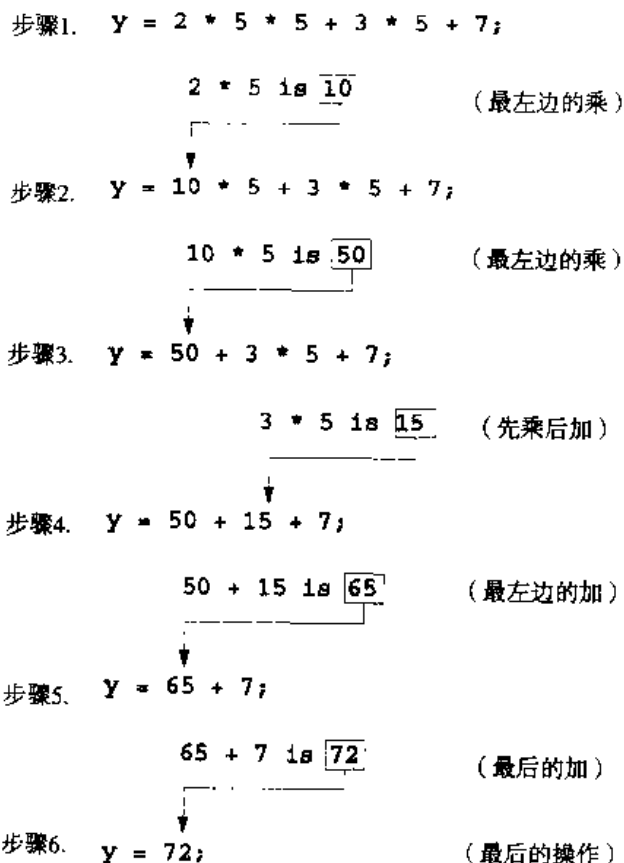


图 1.11 计算一个二次多项式的运算符优先级顺序

在代数中,为了使一个表达式看起来更清晰,可以在其中加入一些不必要的括号。这些不必要的括号也称为冗余括号。例如,上述二次多项式的表达式可以表示为:

$$y = (a * x * x) + (b * x) + c;$$

## 1.20 条件判断:相等运算符和关系运算符

这一节介绍Java中if结构的一个简单版本,程序利用它来按照一些条件的真假值(true或false)进行判断。如果条件满足,即条件为true,则将执行if结构体中的语句。如果条件不满足,即条件为false,结构体中的语句则不会执行。我们将很快看到这样的例子。

if结构中的条件可以使用相等运算符和关系运算符来构造,如图1.12所示。关系运算符拥有同样的优先级,按从左到右的顺序进行结合。相等运算符也有同样的优先级,但是它们的优先级低于关系运算符。相等运算符同样按从左到右的顺序进行结合。

代数中标准的相等运算符和关系运算符	Java 的相等运算符和关系运算符	Java 条件的例子	Java 条件的含义
<b>相等运算符</b>			
=	==	x == y	x 等于 y
≠	!=	x != y	x 不等于 y
<b>关系运算符</b>			
>	>	x > y	x 大于 y
<	<	x < y	x 小于 y
≥	>=	x >= y	x 大于等于 y
≤	<=	x <= y	x 小于等于 y

图 1.12 相等运算符和关系运算符

### 常见编程错误 1.7

在 ==、!=、<=、>= 这样的运算符中间如果包含空格,则会导致语法错误。

### 常见编程错误 1.8

如果将运算符 !=、>=、<= 分别反转成 =!、=>、=<,则将引发语法错误。

### 常见编程错误 1.9

千万不要将相等运算符“==”同赋值运算符“=”混淆。相等运算符应读为“等于”,而赋值运算符应读为“取得”或“得到……的值”。一些人喜欢将相等运算符读为“双等”。

下面的例子使用了6个if语句,对用户在本文本字段中输入的两个数据进行比较。如果这些if语句中的任意条件得到满足,那么将执行同那个if语句相关联的drawString语句。当用户在第二个文本字段中按下回车键时,文本字段中的数据将读进变量num1和num2中。这些比较将在paint方法中进行。图1.13中给出了程序和它的4个输出实例。

注意图1.13中的程序使用了两个单独的TextField对象来接收输入的两个整数。当用户在第二个文本字段中按下回车键时,将调用action方法,然后将执行第55行的if语句。下面这一行:

```
if ( event.target == input2 ) {
```

判断用户是否在第二个文本字段中按下了回车键。记住,当用户在任意一个文本字段中按下回车键后,都将产生一个事件并调用action方法。包含大量信息的Event对象将传递给action,用户可以根

据这些信息来处理这个事件。事件中最重要信息是 `target`。`target` 是图形用户界面的组件，用户通过它来产生事件。`if` 结构中的条件 “`event.target == input2`” 是在询问，即事件的目标是 `input2` 这个文本字段吗？如果上述条件为 `true`，那么将执行这个 `if` 结构体。结构体从第 55 行由一个左花括号 ( `{` ) 开始，在第 59 行以一个右花括号 ( `}` ) 结束。`if` 结构体中如果有多于一条语句出现，我们将使用花括号括起来。下列语句：

```
num1 = Integer.parseInt( input1.getText() );
num2 = Integer.parseInt( input2.getText() );
```

将读出用户在文本字段 `input1` 和 `input2` 中输入的文本，然后转换为整数，并将它们置于变量 `num1` 和 `num2` 中。`TextField` 类中的方法 `getText` 用来从文本字段中读出文本。当执行 `input1.getText()` 之后，程序将得到文本字段 `input1` 中的文本。同样，一旦 `input2.getText()` 执行完毕，文本字段 `input2` 中的内容也由程序获得。

在最初的两个程序中，我们使用了 `paint` 方法在屏幕上显示信息。请记住 `paint` 方法的特殊性，当 `applet` 执行时就会自动调用。尽管如此，在本例中，当用户在文本字段中输入新的值并在文本字段 `input2` 中按下回车键以后，我们希望可以在屏幕上显示不同的信息。下列语句：

```
repaint();
```

用来调用 `paint` 方法。`repaint` 方法是定义 `Comparison` 类时从 `Applet` 类继承的众多方法中的一个，调用后首先激活（调用）另一个继承的称为 `update` 的方法。方法 `update` 将消除原来 `paint` 方法在屏幕上显示的所有信息，然后再次调用 `paint` 方法。

```
1 //using if statements, relational
2 //operators, and equality operators
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class Comparison extends Applet {
7     Label prompt1;// prompt user to input first value
8     TextField input1;// input first value here
9     Label prompt2;// prompt user to input second value
10    TextField input2; // input second value here
11    int num1, num2; // store input values
12
13    // setup the graphical user interface components
14    // and initialize variables
15    public void init ()
16    {
17        prompt1 = new Label ( "Enter an integer" );
18        input1 = new TextField ( 10 );
19        prompt2 =
20            new Label ( " Enter an integer and press Enter " ) ;
21        input2 = new TextField ( 10 );
22        add ( prompt1 ) ; // put prompt1 on applet
23        add ( input1 ) ; // put input1 on applet
24        add ( prompt2 ) ; //put prompt2 on applet
25        add ( input2 ) ; // put input2 on applet
26    }
27
28    // display the results
29    public void paint ( Graphics g )
30    {
```

```

31         g.drawString ( "The comparison results are:", 70, 75 ) ;
32
33         if ( num1 == num2 )
34             g.drawString ( num1 + " == " + num2, 100, 90 );
35
36         if ( num1 != num2 )
37             g.drawString ( num1 + " != " + num2, 100, 105 ) ;
38
39         if ( num1 < num2 )
40             g.drawString ( num1 + " < " + num2, 100, 120 ) ;
41
42         if ( num1 > num2 )
43             g.drawString ( num1 + " > " + num2, 100, 135 ) ;
44
45         if ( num1 <= num2 )
46             g.drawString ( num1 + " < = " + num2, 100, 150 ) ;
47
48         if ( num1 >= num2 )
49             g.drawString ( num1 + " > = " + num2, 100, 165 ) ;
50     }
51
52     // process user 's action on the input2 text field
53     public boolean action ( Event event, Object o )
54     {
55         if ( event.target == input2 ){
56             num1 = Integer.parseInt ( input1.getText () ) ;
57             num2 = Integer.parseInt ( input2.getText () ) ;
58             repaint () ;
59         }
60
61         return true;// indicates user 's action was processed
62     }
63 }

```

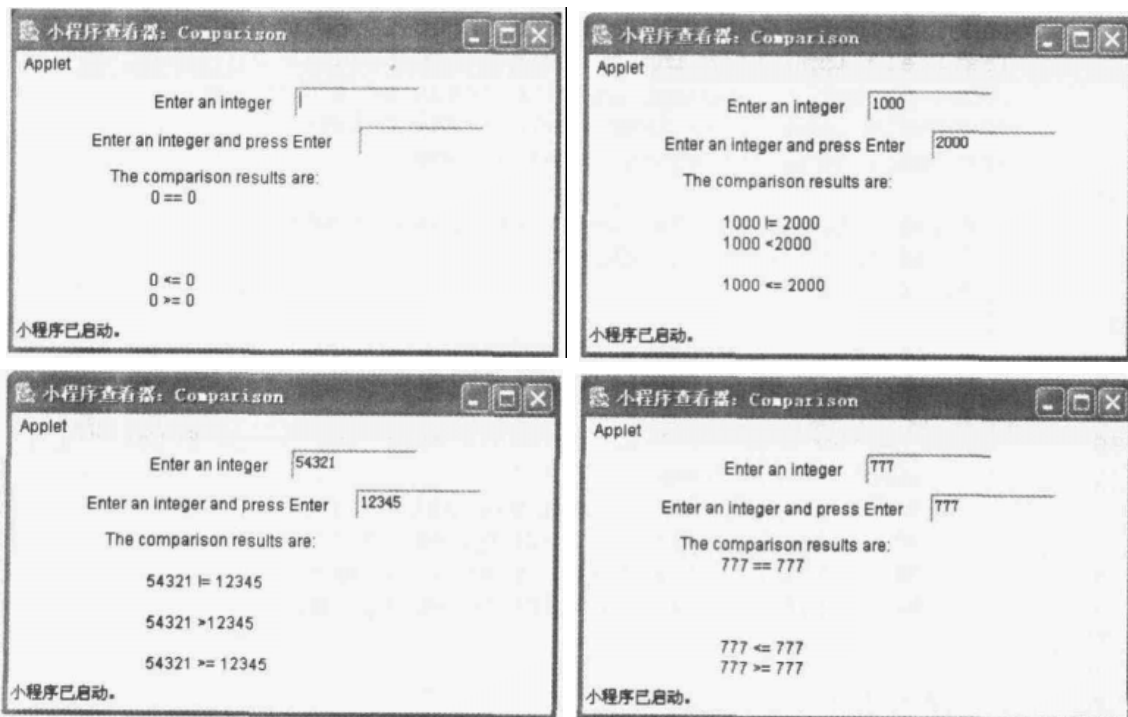


图 1.13 使用相等和关系运算符



paint 中第33行的if结构:

```
if( num1 == num2 )  
    g.drawString( num1 + "==" + num2 ,100, 90 );
```

比较变量 num1 和 num2 是否相等。如果相等, drawString 方法将在屏幕上显示表达式 “num1 + “==” + num2” 的结果。这个表达式看起来十分有趣, 因为它将一个整数同一个字符串相加, 然后把结果同另一个整数相加。Java 有一个 “+” 运算符的特殊版本, 它可以将一个字符串和另一种数据类型的值 (包括其他字符串) 加在一起, 结果通常是一个新字符串。如果我们假设 num1 和 num2 都拥有值 1000, 那么这个表达式的计算过程如下: 首先将 num1 转换为一个字符串, 然后同 “==” 相加, 结果为字符串 “1000==”。接着, 将 num2 转换为一个字符串, 再同 “1000==” 加在一起, 结果是字符串 “1000==1000”。这个字符串将显示在屏幕上。注意将整数 num1 和 num2 自动转换为字符串的操作, 仅当它们同字符串 “==” 相加时发生。字符串之间的相加过程称为字符串的连接, 我们将在第8章中讨论。

#### 常见编程错误 1.10

将字符串连接操作中的 “+” 运算符同加法运算中的 “+” 运算符相混淆将导致奇怪的结果。例如, 假设 y 的值为 5, Java 语句 “g.drawString(“y+2=”+y+2,30,30);” 将显示 “y+2=52” 而不是 “y+2=7”。这是因为 y 的值同 2 相连接, 而不是将它们进行数学上的相加。

#### 常见编程错误 1.11

使用运算符 “+” 将值相加后再用 drawString 方法显示在屏幕上时, 如果被加的值中没有字符串出现, 那么将导致一个语法错误。为了将其他类型的数据自动转换为字符串, 至少要在 “+” 运算符的两边出现一个字符串。

#### 常见编程错误 1.12

将一个 if 结构的条件语句 “if(x==1) ...,” 中的 “==” 替换为 “if(x=1) ...,” 语句中的 “=”, 则将引发一个语法错误。注意, 我们在 if 语句中采用了锯齿型的缩进方法, 这样可以增加程序的可读性。

#### 编程技巧 1.13

if 结构体的缩进可以突出它的结构并且增加程序的可读性。

#### 编程技巧 1.14

一个程序中应该包含不止一行语句。

#### 常见编程错误 1.13

在 if 结构的条件之后的右括号后面紧跟一个分号常常是一个错误。这个分号将会使这个 if 结构体变成空的, 因此无论条件是否满足, 这个 if 语句都不会执行任何操作。更糟糕的是, 原来这个 if 结构体会变成一个顺序结构, 也就是无论条件是否满足, 这一部分语句都将执行。

注意图 1.13 中空格的使用。在 Java 语句中, 空白字符诸如 tab、换行和空格通常是由编译器忽略的。所以, 一条语句可以按照程序员的喜好, 使用空白字符将其分成好几行。但是将标识符和字符串分开是错误的。理想状态下的语句应该总是较短的, 但事实上并非总是如此。

#### 编程技巧 1.15

一条过长的语句应该分成数行。如果一条语句不得被分成几行, 那么应选择有意义的断点, 例如在有逗号的情况下选择在逗号之后, 或者在一个表达式的某个运算符的后面。如果将一条语句分成数行, 最好在每个子行前面缩进。

图1.14描述了本章介绍的运算符的优先级。按从上至下的顺序,运算符优先级将逐级递减。注意所有这些运算符,除赋值运算符“=”外,都是按从左到右的原则结合的。加法是左结合的,因此表达式“ $x+y+z$ ”同“ $(x+y)+z$ ”是按照同样的顺序执行的。赋值运算符从右向左结合,所以表达式“ $x=y=0$ ”同“ $x=(y=0)$ ”是按照同样的顺序执行的。该运算首先将 $y$ 置为0,然后将这个赋值语句的结果(0)赋给 $x$ 。

#### 编程技巧 1.16

在编写一个包含很多运算符的表达式时,请参考以下运算符的优先级顺序表。确信表达式中的运算符按照我们所设想的顺序进行运算。如果在一个复杂的表达式中不能确定运算符的运算顺序,那么可以像在代数表达式中一样使用括号来强制它们的顺序。注意,一些像赋值运算符这样的运算符是从右到左结合而不是从左到右结合。

运算符	结合顺序	类型
()	从左到右	括号
* / %	从左到右	倍数
+ -	从左到右	加减
< <= > >=	从左到右	条件关系
== !=	从左到右	相等关系
=	从右到左	赋值

图 1.14 至今为止所讨论的运算符优先级的关系及结合顺序

我们已经介绍了Java的许多重要特性,其中包括如何在屏幕上输出数据、从键盘读取数据、进行计算以及进行条件判断。在第2章中,当我们介绍结构化程序设计时将使用这些技术,读者将对缩进技术更加熟悉。我们将学习如何设定及调整语句的执行顺序——这样的顺序通常称为控制流程。

## 小结

- 构成一台计算机系统的不同设备(比如键盘、显示屏、磁盘、内存和处理单元)称为硬件。
- 在计算机上运行的程序称为软件。
- 操作系统是一种软件系统,它使用户可以更方便地使用计算机,并且能使计算机表现出最好的性能。
- 分布式计算是指对一件事务的处理不再集中在某一台中心计算机,而是通过网络将它分布到许多不同的设备上。
- 一些机器称为文件服务器,它们的作用是为程序和数据提供巨大的存储空间;一些机器称为客户,它们通过网络使用文件服务器上的有关信息。这就是我们所说的客户/服务器结构。
- 任何计算机都只能直接理解它自己的机器语言。
- 机器语言通常由数字串组成(最终也是0和1的组合),它执行计算机中最基本的操作。该语言依赖于特定的机器,也就是一种机器语言只能用在某一特定类型的计算机上。
- 类似英文的缩略语构成了汇编语言的基础。称为汇编器的翻译程序可以用来在计算机上快速地将汇编程序转换为机器语言。
- 将高级语言编写的程序转换为机器语言的翻译程序称为编译器。高级语言(例如Java)允许程序员采用接近日常英语的指令来编写程序,传统的数学符号同样可以使用。
- 解释器不需要将高级语言编写的程序编译成机器语言,它们可以直接执行高级语言。采用解

释器执行程序的一个好处是可以节省编译所耗费的时间。

- 尽管编译好的程序执行速度远远快于在解释器中执行的程序,但是在那些由于增加新的性能或者正在调试错误而需要频繁重编译的开发环境中,解释器的应用仍然相当广泛。然而,一旦程序开发完成,发行的将是一个运行更快的编译好的版本。
- 采用这种较为严格的结构化编程方法写出的程序同非结构化程序相比更加清晰,更易于调试和跟踪,并且更易于修改。
- 通常,一个Java系统包含以下几部分:一个环境,Java语言本身,Java应用程序接口(API),以及各种Java类库。
- 像“除数为零”这样的在程序运行时发生的错误称为运行时错误或者执行时错误。
- 致命的运行时错误将导致程序在完成任务之前立即终止。非致命错误可能使程序继续运行,但常常导致不正确的结果。
- 对象本质上是一些可重用的软件组件,它们模仿了现实世界中的事物。
- Java applet是在Netscape的Navigator或Sun的HotJava这样的WWW浏览器上运行的程序。
- 程序员在程序中加入注释来使程序文档化,注释增加了程序的可读性。注释还可以帮助其他人阅读和理解读者的程序。注释在程序执行时不会导致计算机执行任何动作。Java编译器将忽略注释,并且不会产生任何字节码。
- 采用“//”开始的注释称为单行注释,因为这个注释的作用在行尾结束。
- Java包括许多预先定义的类,并且在磁盘上将相关的类放入一个目录中,我们称之为软件包。这些软件包又称为Java类库或者Java应用程序编程接口(Java API)。
- import语句用来载入编译一个Java程序所需要的类。
- 载入Graphics类的目的是程序可以在屏幕上显示信息。
- 关键字class在Java中引入了一个类的定义。
- 关键字extends紧跟在类的名字后面,指出我们的新类从哪里继承而来。
- 所有的Java applet必须从Applet类中继承而来。
- 类用来在程序执行时在内存中实例化(或创建)一个对象。一个对象是内存中的一个区域,其中存储程序使用的信息。
- 在Java中,主要的Applet类(即从Applet中继承而来的)总是一个public(公有)类型的类。
- 在将applet存入文件时,这个applet的类名将作为文件名的一部分(扩展名必须用.java)。
- 左花括号({)表明一个类定义的开始,对应的右花括号(})表明类定义的结束。
- paint方法在一个applet执行时自动调用,用来在屏幕上显示信息。
- 关键字void表明一个方法将履行某一任务,但是当该方法结束时并不返回值。
- 方法从参数表中接收完成任务所需的信息。
- 左花括号表明方法的结构体的开始,相应的右花括号表明这个方法的结构体的结束。
- 每条语句都必须以分号结束(也称之为语句结束符)。
- 图形坐标的第一个坐标是x坐标,代表屏幕上从applet左上角自左向右计数的像素个数;第二个坐标是y坐标,代表从上向下计数的像素个数。大多数Graphics类中的画图方法要求至少一个坐标集作为参数,以便告诉在Java applet上从何处开始画图。
- 必须创建一个HTML(超文本标识语言)文档来载入一个applet,以便浏览器可以执行它。
- 当编译Java程序时,将每一个类编译成一个单独的文件,并以类名带.class后缀作为文件名。
- HTML的applet标记是专为Java applet使用的。它告诉浏览器载入某一特定的以.class为后缀

的 Java applet，并规定它的大小和起始位置

- appletviewer 是一个测试 Java applet 的程序。appletviewer 需要一个 HTML 文件来载入 Java applet。
- GUI 组件可以帮助程序的用户输入数据和输出数据。
- 变量由原始的（或称为内建的）数据类型所定义，例如 int 类型。Java 程序中的每一个数据都将看成是一个对象，但由原始数据类型定义的变量除外。
- TextField（文本字段）对象用来从键盘接收用户的输入并把信息显示在屏幕上。
- Label（标签）包含一条将显示在屏幕上的字符串，Label 通常同屏幕上另一种图形用户界面的组件相联系。
- 所有变量和引用都必须在程序使用之前声明一个名字及其类型。
- 标识符是指这样的一串字符，它包含字母、数字、下划线（\_）以及美元符号（\$），并且不以数字开头。Java 允许任何长度的标识符。Java 是区分大小写的，即大写和小写字符是不同的。
- 无论如何都应该在方法使用变量之前定义该变量。
- init 方法通常在一个 applet 中初始化这个 applet 用到的变量和引用。init 方法在一个 applet 开始执行时将自动调用，并且它是首先调用的方法。
- applet 通常按顺序调用三个方法——init、start 和 paint。如果没有为这三个方法中的某一个（或多个）进行自己的定义，就将从 Applet 类中得到一个“免费”的版本，但是它（它们）不执行任何操作。
- 一个 prompt（提示）对象告诉用户去完成一个特定的操作。
- 运算符“=”称为二元运算符，因为它有两个操作数。
- add 方法用来将 GUI 组件放在一个 applet 中。
- 只有类的对象必须采用动态的方式（new）来申请空间。原始类型的变量由 Java 自动分配空间。
- 所有在用户和 GUI 组件之间的交互操作都将产生事件。
- 用户同 GUI 组件之间的交互操作过程称为事件驱动的编程。
- 当用户在文本字段中按下回车键时将调用 action 方法，action 方法中的 Object 参数包含文本字段中的内容。
- action 方法的 Event 参数包含事件的目标 target，即用户在哪个图形用户界面的组件上产生了这一事件。
- 任何类型的对象都可以采用 toString 方法转化为 String 类型。一个 String 是一个可以存储字符串的对象。
- Integer.parseInt 方法的作用是将它的 String 类型的参数转化为一个整数。
- TextField 类中的方法 setText 将文本设置在指定的文本字段中。
- 大多数计算在赋值语句中出现。
- 状态栏在 appletviewer 或其他浏览器的底部。
- Applet 类的方法 showStatus 在状态栏上显示一个 String 类型。
- 每一个变量都有名称、类型、存储长度和值。
- 无论一个内存空间是否已经有一个值，新的值将取代其中原有的值，原有的值将消失。
- 当从一个内存空间读出一个值时，并不破坏原来的内容。
- 算术运算符都是二元运算符。

- 整除的结果也为整数。
- 取模运算符“%”可以得到整除之后的余数。
- Java 在计算算术表达式时按照运算符优先级顺序进行操作，它们同传统代数中的约定是相同的。可以使用括号强制改变计算顺序。
- 程序使用if结构来按照一些条件的真假值进行判断。如果条件满足，即条件为true，就执行if结构体中的语句。如果条件不满足，即条件为false，则不执行结构体中的语句。
- if结构中的条件可以使用相等运算符和关系运算符来构造。
- TextField类中的方法getText用来从文本字段中读取文字。
- repaint方法调用update方法，清除原来屏幕上显示的所有信息，然后再次调用paint方法。
- Java有一个“+”运算符的特殊版本，它可以将一个字符串和另一种数据类型的值（包括其他字符串）加在一起，结果通常是一个新的字符串。字符串之间的相加过程称为字符串的连接。
- 诸如tab、换行和空格之类的空白字符通常是由编译器忽略的。所以，一条语句可以按照程序员的喜好来用空白字符分成好几行。但是将标识符和字符串用空白字符分开则是错误的。

## 术语

abstraction 抽象

action 动作

action method action 方法

add method add 方法

ANSI C

applet Java 小程序

Applet class Applet 类

appletviewer appletviewer 命令

application 应用程序

arithmetic and logic unit(ALU)

算术及逻辑单元 (ALU)

arithmetic operators 算术运算符

assembly language 汇编语言

assignment operator(=) 赋值运算符 (=)

associativity of operators 运算符的结合律

base class 基类

binary operators 二元运算符

body of a method 方法的结构体

boolean primitive type boolean 原始类型

bytecodes 字节码

C

C standard library 标准C库

C++

C++ class libraries C++ 类库

case sensitive 大小写敏感

central processing unit(CPU) 中央处理器 (CPU)

character string 字符串

class 类

.class file extension .class 文件扩展名

class keyword 关键字 class

client/server computing 客户/服务器计算

comment(//) 注释 (//)

compile error 编译错误

compile-time error 编译时错误

compiler 编译器

computer 计算机

computer program 计算机程序

condition 条件

CPU

decision 判断

declaration 声明

derived class 派生类

distributed computing 分布式计算

drawString method drawString 方法

editor 编辑器

equality operators 相等运算符

- == “is equal to” == “等于”
- != “is not equal to” != “不等于”
- Event class Event 类
- execution-time error 运行时错误
- extends keyword 关键字 extends
- false value false 值
- fatal error 致命错误
- file server 文件服务器
- flow of control 控制流
- getText method getText 方法
- Graphics class Graphics 类
- hardware 硬件
- high-level language 高级语言
- HTML (Hypertext Markup Language)  
HTML (超文本标记语言)
- identifier 标识符
- if structure if 结构
- import statement import 语句
- information hiding 信息隐藏
- inheritance 继承
- input device 输入设备
- input/output(I/O) 输入/输出 (I/O)
- int primitive type int 原始类型
- integer(int) 整型 (int)
- Integer class Integer 类
- integer division 整除
- interface 接口
- interpreter 解释器
- Java
- Java Developer's Kit(JDK)  
Java 开发工具集 (JDK)
- .java file extension .java 文件扩展名
- Java Virtual Machine Java 虚拟机
- Label class Label 类
- logic error 逻辑错误
- machine dependent 依赖于机器的
- machine independent 不依赖于机器的
- machine language 机器语言
- main
- memory 内存
- memory location 内存位置
- message 消息
- method 方法
- modulus operator(%) 取模运算符 (%)
- multiplication operator(\*) 乘法运算符 (\*)
- multiprocessor 多处理器
- multiprogramming 多道程序
- multitasking 多任务
- multithreading 多线程
- Netscape browser Netscape 浏览器
- nested parentheses 嵌套的括号
- nonfatal error 非致命错误
- object 对象
- Object class Object 类
- object-oriented design(OOD)  
面向对象的设计 (OOD)
- object-oriented programming(OOP)  
面向对象的编程 (OOP)
- operand 操作数
- operator 运算符
- output device 输出设备
- paint method paint 方法
- parentheses() 括号
- parseInt method parseInt 方法
- precedence 优先级
- primary memory 主存
- procedural programming 过程化编程
- programming language 编程语言
- public keyword 关键字 public
- reference 引用
- relational operators 关系运算符
- > “is greater than” > “大于”
- < “is less than” < “小于”
- >= “is greater than or equal to” >= “大于等于”
- <= “is less than or equal to” <= “小于等于”
- repaint method repaint 方法
- reserved words 保留字
- right-to-left associativity 从右到左的结合律
- rules of operator precedence 运算符优先级规则
- run-time error 运行时错误
- semicolon(;)statement terminator 分号——语句  
终止符

software 软件	update method update 方法
statement 语句	variable 变量
string 字符串	variable name 变量名
structured programming 结构化编程	variable value 变量值
syntax error 语法错误	void keyword 关键字 void
target instance variable 实例变量 target	write space character 空白字符
Textfield class Textfield 类	World Wide Web WWW (万维网)
true value true 值	

## 自测练习

### 1.1 填空:

- 将个人计算机推广开来的公司是\_\_\_\_\_。
- 使得个人计算在商业和工业领域确立地位的计算机是\_\_\_\_\_。
- 计算机用来处理数据的指令集称为\_\_\_\_\_。
- 计算机上的6个逻辑单元为\_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_和\_\_\_\_\_。
- 本章讨论的三种语言是\_\_\_\_\_, \_\_\_\_\_和\_\_\_\_\_。
- 将高级语言的程序转换为机器语言程序的程序称为\_\_\_\_\_。
- 由Wirth教授开发出来的用于大学进行结构化程序设计教学的语言是\_\_\_\_\_。
- 美国国防部开发的Ada语言具有一种称为\_\_\_\_\_的能力, 它可以使程序员写出并发的多个任务。

### 1.2 填空:

- Java 开发工具集中的\_\_\_\_\_命令用来执行Java applet。
- Java 开发工具集中的\_\_\_\_\_命令用来执行Java 应用程序。
- Java 开发工具集中的\_\_\_\_\_命令用来编译Java 程序。
- \_\_\_\_\_文件用来激活一个Java applet。
- Java 程序文件必须以\_\_\_\_\_作为文件名后缀。
- 当编译一个Java 程序时, 编译器产生的编译文件后缀名为\_\_\_\_\_。
- Java 编译器产生的文件包含\_\_\_\_\_, 它用来解释执行一个Java applet或应用程序。

### 1.3 填空:

- 方法的结构体由\_\_\_\_\_开始, 以\_\_\_\_\_结束。
- 一条语句以\_\_\_\_\_结束。
- \_\_\_\_\_结构用来进行条件判断。

### 1.4 判断下列语句是否正确。如果不正确, 请解释原因。

- 当程序执行时, 注释用来在屏幕上显示//符号之后的内容。
- 一个方法中用到的所有变量必须在使用之前进行声明。
- 当定义一个变量时, 必须指出它的类型。
- Java 认为变量number和NuMbEr是相同的。
- 声明可以出现在Java 的方法体中的任何位置。

- f) 整除 (%) 运算符仅用于整数操作数。
- g) 算术运算符 \*、/、%、+ 和 - 都拥有同样的优先级
- 1.5 按照下述要求写出 Java 语句:
- 声明变量 `c`、`thisIsAVariable`、`q76354` 和 `number`，类型是 `int`。
  - 提示用户输入一个整数，声明一个 `Label` 引用，创建一个 `Label` 对象，然后说明如何在 applet 上放置这个 `Label`。
  - 从键盘上读入一个整数并且将它放在整数变量 `age` 中。假设 `TextField` 调用已有的 `input`，并且 `action` 方法的 `Object` 类型参数 `o` 包括用户输入的内容。
  - 如果变量 `number` 不等于 7，请用 `paint` 方法在 (10,10) 的位置上显示 “The variable number is not equal to 7.”。假设 `Graphics` 的对象 `g` 作为 `paint` 方法的参数。
  - 在 `paint` 方法中的一行显示信息 “This is a Java program”。假设 `Graphics` 的对象 `g` 作为 `paint` 方法的参数，可以自己选择起始坐标。
  - 使用 Java 语言的两个显示行来显示 “This is a Java program”。假设 `Graphics` 的对象 `g` 作为 `paint` 方法的参数，可以自己选择起始坐标。
- 1.6 找出并改正下列句中的错误:
- ```
if ( c < 7 ) ;
    g.drawString( " c is less than 7 " , 25 , 25 );
```
  - ```
if ( c ==> 7)
    g.drawString( "c is equal to or greater than 7" , 25 , 25 );
```

## 自测练习答案

- 1.1 a) Apple。b) IBM 个人计算机。c) 程序 d) 输入单元，输出单元，内存单元，算术和逻辑单元，中央处理单元，二级存储单元。e) 机器语言，汇编语言，高级语言。f) 编译器。g) Pascal。h) 多任务。
- 1.2 a) appletviewer。b) java。c) javac。d) HTML。e) java。f) .class。g) 字节码。
- 1.3 a) 左花括号，右花括号。b) 分号。c) if。
- 1.4 a) 不正确。注释在程序执行时不引发任何动作，它们用于使程序文档化并增加程序的可读性。
- b) 正确。
- c) 正确。
- d) 不正确。Java 是大小写敏感的，所以这两个变量是不同的。
- e) 正确。
- f) 正确。
- g) 不正确。运算符 \*、/、% 拥有同样的优先级，运算符 +、- 同在一个更低的优先级上。
- 1.5 a) `int c, thisIsAVariable, q76354, number ;`
- b) `Label prompt ;`  
`prompt = new Label( "Enter an integer");`  
`add( prompt );`
- c) `age = Integer.parseInt( o.toString() );`
- d) `if ( number != 7 )`



```
g.drawString( "The variable number is not equal to 7.", 10 , 10 ) ;  
e) g.drawString( "This is a Java program", 10 , 10 );  
f) g.drawString( "This is a Java" , 10 , 10) ;  
g.drawString( "program", 10 , 25 );
```

- 1.6 a) 错误。分号紧跟在if语句条件表达式的右括号后面。

改正：去掉右括号后面的分号。注意，无论if语句中的条件是否满足，这个错误的结果将导致输出语句都得到执行。分号紧跟在右括号后面表示一个空语句（一个不做任何操作的语句）。我们将在下一章学习有关空语句的更多知识。

- b) 错误。关系运算符==的用法错误。

改正：将==改成>=。

## 练习

- 1.7 将下述条目分类成硬件和软件：

- a) CPU
- b) Java 编译器
- c) ALU
- d) Java 解释器
- e) 输入单元
- f) 一个编辑程序

- 1.8 为什么一般程序员编写程序时更愿意采用一个不依赖于硬件的语言？为什么编写特定类型的程序时一种依赖于硬件的语言更加合适？

- 1.9 填空：

- a) 计算机使用哪一种逻辑单元从外界接收信息？\_\_\_\_\_。
- b) 命令计算机解决特定问题的过程称为\_\_\_\_\_。
- c) 哪种类型的计算机语言使用类似英文缩写的助记符表达机器语言指令？\_\_\_\_\_。
- d) 哪种类型的逻辑单元用来将计算机处理完的信息发往其他各种设备，以便这些信息在本机之外得到使用？\_\_\_\_\_。
- e) 计算机使用哪种逻辑单元保留信息？\_\_\_\_\_。
- f) 计算机使用哪种逻辑单元进行计算？\_\_\_\_\_。
- g) 计算机使用哪种逻辑单元进行逻辑判断？\_\_\_\_\_。
- h) 对于程序员来说，使用哪种级别的计算机语言编写程序将更快更容易？\_\_\_\_\_。
- i) 计算机能直接理解的惟一语言是\_\_\_\_\_。
- j) 哪种计算机逻辑单元用来协调其他逻辑单元的活动？\_\_\_\_\_。

- 1.10 填空：

- a) \_\_\_\_\_ 用来使程序文档化并增加程序的可读性。
- b) 一个\_\_\_\_\_类的对象能够接收用户从键盘输入的信息，或者在一个 applet 上显示信息。
- c) 用来进行条件判断的 Java 语句为\_\_\_\_\_。
- d) 计算通常都在\_\_\_\_\_语句中完成。

- e) 一个\_\_\_\_\_类的对象用来显示文本, 而此文本常常同这个 applet 中的另一个 GUI 组件相关联。
- 1.11 按下述要求写出 Java 语句:
- a) 使用一个 Label 显示信息 "Enter two number".
  - b) 将变量 b 和 c 的积赋给变量 a
  - c) 说明一个用来进行工资计算的例程 (即用文字对程序加以注释)。
- 1.12 判断下列语句是否正确。如果不正确, 请解释原因
- a) Java 运算符都是从左到右进行计算的。
  - b) 下述标识符都是合法变量名: under\_bar\_, m928134, t5, j7, hre\_salses, his\_account\_total, a, b, c, z, z2。
  - c) 一个没有括号的合法 Java 算术表达式是按照从左到右的顺序进行计算的。
  - d) 下述均为非法变量名: 3g, 87, 67h2, h22, 2h
- 1.13 填空:
- a) 同乘法拥有相同优先级的算术运算符是\_\_\_\_\_。
  - b) 如果一个算术表达式中有嵌套的括号, 那么首先计算哪对括号中的内容? \_\_\_\_\_。
  - c) 在一个程序执行过程中, 计算机内存中某一位置的内容将随着程序的执行而改变, 我们将这个内存位置称为\_\_\_\_\_。
- 1.14 下述语句在执行时将输出什么结果? 如果没有输出请写明 "没有"。设  $x=2$  和  $y=3$ 。
- a) `g.drawString( Integer.toString (x) , 25 , 25);`
  - b) `g.drawString( Integer.toString (x + x), 25 , 25);`
  - c) `g.drawString( "x = " , 25 , 25);`
  - d) `g.drawString( "x = " + x, 25 , 25);`
  - e) `g.drawString( (x + y) + " = " + (y + x), 25 , 25);`
  - f) `z = x + y;`
- 1.15 下述 Java 语句中, 哪个变量的值将会改变?
- a) `p = i + j + k + 7;`
  - b) `g.drawString( "variables values destroyed" , 25 , 25);`
  - c) `g.drawString( "a = 5" , 25 , 25);`
- 1.16 给定  $y = ax^3 + 7$ , 下述语句的哪一个正确地表述了这一表达式?
- a) `y = a * x * x * x + 7;`
  - b) `y = a * x * x * (x + 7);`
  - c) `y = (a * x) * x * (x + 7);`
  - d) `y = (a * x) * x * x + 7;`
  - e) `y = a * (x * x * x) + 7;`
  - f) `y = a * x * (x * x + 7);`
- 1.17 说明下面的 Java 语句中运算符的计算顺序, 并给出语句执行完后 x 的值。
- a) `x = 7 + 3 * 6 / 2 - 1;`
  - b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
  - c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`
- 1.18 编写一个 Java applet, 要求用户输入两个数, 然后从用户那里接收这两个数, 打印它们的和、积、差及商。请使用图 1.13 所示的图形用户界面技术。

- 1.19 编写一个Java applet, 要求用户输入两个整数, 然后从用户那里接收这两个数, 将它们当中较大的一个和字符串“is larger”一起显示在 applet 的状态栏上。如果两个数相等, 则显示信息 “These numbers are equal”。请使用图 1.13 所示的图形用户界面技术。
- 1.20 编写一个Java applet, 从键盘输入三个整数, 然后打印它们的和、平均值、积、最小数的值以及最大数的值。请使用图 1.13 所示的图形用户界面技术。
- 1.21 编写一个Java applet, 输入一个圆的半径, 然后输出它的直径、周长和面积,  $\pi$  的值为 3.141 59。请使用图 1.6 所示的图形用户界面技术。
- 1.22 编写一个Java 程序, 输出下面所示的矩形、椭圆、箭头和菱形:

```

*****          ***          *          *
*              *      *      *      ****      *      *
*              *      *      *      *****     *      *
*              *      *      *      *            *      *
*              *      *      *      *            *      *
*              *      *      *      *            *      *
*              *      *      *      *            *      *
*              *      *      *      *            *      *
*****          ***          *          *

```

- 1.23 Java 的 Graphics 类提供了许多有用的绘图方法, 我们将在第 9 章中详细讨论。现在编写一个Java applet, 它使用方法 paint 中的下述语句画出一个矩形:

```
g.drawRect( 10 , 10 , 50 , 50 );
```

Graphics 类中的方法 drawRect 带有 4 个参数。前面两个是这个矩形左上角的坐标, 第三个参数是矩形按像素计算的宽, 第 4 个参数是矩形按像素计算的高。

- 1.24 修改练习 1.23 的程序, 从而输出不同大小的矩形。
- 1.25 Graphics 类包含一个 drawOval 方法, 它同方法 drawRect 一样带有 4 个参数。尽管如此, drawOval 的参数实际上指定了这个椭圆的外切矩形。外切矩形的边同椭圆的边相切。编写一个Java applet, 使用相同的 4 个参数画一个椭圆和一个矩形, 椭圆同矩形在每一条边的中央相切。
- 1.26 修改练习 1.25 的结果, 使其输出不同大小和形状的椭圆。
- 1.27 下述代码的输出结果是什么?

```

g.drawString( "*" , 25, 25 );
g.drawString( "****", 25 , 55 );
g.drawString( "*****", 25, 85 );
g.drawString( "*****", 25 , 70 );
g.drawString( "****" , 25 , 40 );

```

- 1.28 编写一个Java applet, 读取一组整数, 输出它们之中最大和最小的值。只能使用本章学到的程序设计技术。
- 1.29 编写一个Java applet, 读取一个整数, 输出并判断它的奇偶性 (提示: 使用取模运算符。一个偶数是 2 的倍数, 任何 2 的倍数除以 2 时其余数为 0)。
- 1.30 编写一个Java applet, 读取两个整数, 输出并判断第一个整数是否为第二个整数的倍数。(提示: 使用取模运算符。)

- 1.31 编写一个 Java applet，显示下面所示的图案：

```

*****
*****
*****
*****
*****
*****
*****
*****
*****
*****

```

- 1.32 区别什么是致命错误，什么是非致命错误。为什么有时候一些程序员宁愿遇到一个致命错误而不是非致命错误？

- 1.33 在本章，我们学习了整数和它的类型 `int`。Java 中可以用到各种大写字符、小写字符和相当数量的特殊字符。每一个字符都和一个整数相对应，一台计算机使用的字符集合以及这些字符对应的整数构成了这台机器的字符集。可以简单地将一个字符用单引号括起来以便使用，例如 `'A'`。

可以通过在一个字符的前面加上 `"(int)"` 来获得这个字符所对应的整数值，这称为强制类型转换（我们将在第 2 章讲述这方面的更多知识）。

```
(int) 'A'
```

下面的语句将在一个 applet 的方法 `paint` 中输出一个字符和它对应的整数：

```
g.drawString("The character " + 'A' + " has the value " + (int) 'A',
25 , 25 );
```

当执行这条语句时，它将字符 `A` 和 `65` 这个值（在使用 Unicode 字符集的系统上）作为字符串的一部分显示出来。编写一个 Java applet，将一些大写字符、小写字符和特殊字符及它们对应的整数显示出来。至少显示以下字符：`A, B, C, a, b, c, 0, 1, 2, $, *, +, /`，空格字符。

- 1.34 编写一个 Java applet，输入一个 5 位数，将每个数字分开后在其中分别插入三个空格，然后输出。例如，如果用户键入 `42339`，程序将输出：

```
4    2    3    3    9
```

- 1.35 利用仅在本章学到的一种技术，编写一个 Java applet，计算如下表所示的从 0 到 10 这些数的平方和立方值：

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

## 第2章 控制结构（一）

### 教学目标

- 理解解决问题的基本技巧
- 学会自顶向下、逐步求精的开发算法
- 学会使用 if/else 选择结构来选择执行不同的操作
- 学会使用 while 循环结构来重复地执行程序中的语句
- 理解计数器控制循环和标志控制循环
- 学会使用自增、自减和赋值运算符

### 2.1 简介

在编写解决某个问题的程序之前,最重要的是彻底了解该问题,并精心地计划如何解决这个问题。在编写程序时,同样重要的是要了解各组成模块的类型,并使用已验证过的程序构造原则。我们将在本章和下一章讨论结构化程序设计的理论和方法,这里学到的技术适用于包括Java在内的大多数编程语言。当我们在第8章更进一步地学习面向对象的编程时,就会发现控制结构将有助于创建和操作对象。

### 2.2 算法

任何可计算性问题的解决过程,都可以转化为按指定顺序执行的一系列操作过程。由下列两个部分组成的解决问题的过程称为算法:

1. 可执行的操作
2. 执行操作的顺序

从下面的例子中可以看出,可执行操作的顺序正确与否是很重要的。

考虑下面的“早起算法”,这是一个普通人早晨完成的起床和上班过程:(1)起床;(2)脱掉睡衣;(3)洗澡;(4)穿衣服;(5)吃早餐;(6)开车去上班。

这个执行过程将使这个人处于良好的工作状态。设想一下,同样的步骤如果稍微改变一下顺序会是什么结果:(1)起床;(2)脱掉睡衣;(3)穿衣服;(4)洗澡;(5)吃早餐;(6)开车上班。

如果按这个步骤来执行,这个人就只能湿漉漉地去上班了。在计算机程序中指定语句的执行顺序称为程序控制。我们将在本章和第3章讨论Java的程序控制功能。

### 2.3 伪代码

伪代码是一种可以帮助程序员开发算法的人工非正式语言,它对于开发将转换为Java程序的

结构化部件的算法非常有用。伪代码与英语的结构相似,虽然它不是真正的计算机程序设计语言,但使用起来非常方便,并且十分通俗易懂。

伪代码程序并不能在计算机中实际运行,但是它可以帮助程序设计人员先将程序“设想”出来,然后再使用像 Java 这样的编程语言来编写程序。本章我们将给出几个伪代码程序的例子。

#### 软件工程视点 2.1

在程序设计过程中,通常先用伪代码来“设想”程序,然后再将伪代码程序转换为 Java 程序。

我们这里使用的伪代码是由纯字符组成的,因此程序员可以使用一个编辑器来很方便地编写伪代码程序。可以在需要的时候随时将伪代码程序打印出来。一个精心准备的伪代码程序比较容易转换为相应的 Java 程序。在许多例子中,只需用 Java 语言的相应语句替换伪代码语句,就可以实现这种转换。

伪代码仅由可执行语句组成,可执行语句是指从伪代码转换为 Java 程序后可以执行的语句。声明不是可执行语句。例如,下面这条声明语句:

```
int i;
```

其功能是告诉编译器变量 *i* 的类型,并指示编译器在内存中为该变量保留空间,这个声明在程序运行时并不产生任何诸如输入、输出或计算等动作。有些程序员乐于在伪代码程序的开始列出所用的变量并指出它们的用途

## 2.4 控制结构

一般情况下,程序中的语句将按顺序逐句地执行,这称为顺序执行。我们将要讨论的几种 Java 语句使程序员能够根据自己的需要来指定要执行的下一条语句。可以不按语句顺序进行,这称为控制转换。

20 世纪 60 年代,软件开发组织发现,毫无节制地使用控制转换成为了各种问题的根源,这里主要指的是 `goto` 语句。`goto` 语句使程序员可以将控制转移到程序的很多位置。而结构化编程的观点几乎是“取消 `goto` 语句”,因此 Java 中没有 `goto` 语句。

Bohm 和 Jacopini<sup>①</sup>的研究表明,程序设计可以不使用 `goto` 语句。对于程序员来说,这个时代对他们的挑战是将编程风格转变为“无 `goto` 语句编程”。直到 20 世纪 70 年代,程序员们才开始严肃地对待结构化编程。其结果正像软件开发组织所报告的那样:结构化编程缩短了开发时间,出现了更多按时完成系统设计、完成软件项目而不会超出预算的情况。取得这些成功的关键是结构化的程序更整齐,更易于调试和修改,而且更不容易出现错误。

Bohm 和 Jacopini 的工作表明,所有的程序都可以仅使用三种控制结构来表示,它们分别是顺序结构、选择结构和循环结构。Java 的基本结构是顺序结构,除非特别指明,否则计算机将按 Java 语句的顺序逐句地执行。图 2.1 的流程图展示了一个典型的顺序结构,这个程序将按顺序执行两步计算。

流程图是一种代表整个算法或部分算法的图形。流程图是由具有不同含义的图形组成的,包括矩形、菱形、椭圆形和小圆圈等,这些符号由称为流程线的箭头连接起来。

<sup>①</sup> Bohm, C., G. Jacopini, "Flow Diagrams, Turing Machine, and Languages with Only Two Formation Rules." Communications of the ACM, Vol 9, No. 5, May 1966, pp. 336 ~ 371.

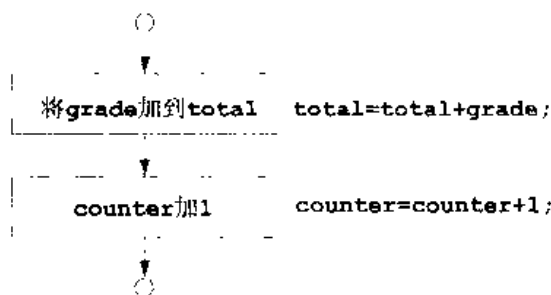


图 2.1 Java 顺序结构的流程图

同伪代码一样，流程图有助于开发和表示算法，不过大多数的开发人员更喜欢伪代码。流程图清楚地显示了控制结构的操作情况，读者应仔细地比较一下每一个用伪代码和流程图表示的控制结构。

再看一下图 2.1 中顺序结构的流程图。我们用矩形符号（或者称为运算符号）表示任意操作，包括计算或输入/输出操作。图中的流程线表示操作执行的顺序：首先将 grade 与 total 相加，然后将 counter 加 1。Java 在顺序结构中不限制操作的个数，我们在后面将会看到，任何可以放置单个语句的地方也可以按顺序放置多个语句。

在绘制一个表示完整算法的流程图时，包含“Begin”的椭圆符号是流程图中的第一个符号；包含“End”的椭圆符号是流程图中的最后一个符号。当只绘制流程图的一部分时（如图 2.1 所示），可以省略椭圆符号，而用小圆圈（也称为连接符）符号代替。

流程图中最重要的是菱形符号，也称为判断符。我们将在下一节介绍菱形符号。

Java 有三种选择结构，我们将在本章和第 3 章中讨论。if 选择结构在条件为 true 时执行一个操作，条件为 false 时则跳过该操作。if/else 选择结构在条件为 true 时执行一个操作，当条件为 false 时执行另一个操作。switch 选择结构（参见第 3 章）是根据表达式的值来执行许多不同操作中的某一个操作。

if 结构称为单选择结构，因为它执行或忽略的是单个操作。if/else 结构称为双选择结构，因为它是在两个操作中选择执行一个操作。switch 结构称为多选择结构，因为它是在许多不同的操作中选择一个。

Java 有三种循环结构：while、do/while 和 for（我们将在第 3 章介绍 do/while 和 for）。if、else、switch、while、do 和 for 都是 Java 的关键字。这些关键字已成为 Java 语言的保留字，以实现不同的功能，例如 Java 的控制结构。关键字是不能用做标识符的，例如变量名。图 2.2 中列出了所有的 Java 关键字。

关键字				
abstract	boolean	break	byte	case
catch	char	class	continue	default
do	double	else	extends	false
final	finally	float	for	if
implements	import	instanceof	int	interface
long	native	new	null	package
private	protected	public	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		
由 Java 保留的关键字，但未使用				
byvalue	cast	const	future	generic
goto	inner	operator	outer	rest
var				

图 2.2 Java 关键字

### 常见编程错误 2.1

使用关键字作为标识符

Java 只有七种控制结构：一种顺序结构、三种选择结构和三种循环结构。每个程序都是由适合于实现算法的多种控制结构组成的。如图 2.1 所示，每个控制结构的流程图都有两个小圆圈符号，一个是控制结构的入口，一个是出口。

使用单入口 / 单出口控制结构来创建程序是比较容易的，只需将一个控制结构的出口与下一个控制结构的入口连接。这与儿童堆积木的方式相似，因此我们称之为控制结构的堆栈。另外还有一种连接控制结构的方法，称为控制结构的嵌套。在 Java 程序的算法中，七种控制结构只有两种连接方式。

## 2.5 if 选择结构

选择结构是从程序的许多操作中选择一个操作来执行。例如，假设考试的及格线是 60 分（满分为 100 分），下列伪代码：

```
If student's grade is greater than or equal to 60
    Print " Passed"
```

其含义是判断条件 “student's grade is greater than or equal to 60” 为 true 还是为 false，如果条件为 true，则打印 “Passed”，然后按顺序 “执行” 下一条伪代码语句（请记住，伪代码并不是真正的编程语言）。如果条件为 false，则忽略打印语句，然后按顺序执行下一条伪代码语句。注意，该选择结构的第二行缩进了几格。这种缩进格式不是必需的，但由于它突出了结构化程序的固有结构，因此这里推荐采用这种格式。Java 编译器将忽略空白字符，例如用于产生缩进格式和垂直间隔的空格、制表符和换行符，使用这些空白字符可增强程序的清晰度。

### 编程技巧 2.1

使用缩进格式会大大提高程序的可读性。我们建议每一层缩进的长度固定为三个空格。

把前面 if 语句的伪代码写成 Java 语句，如下所示：

```
if ( grade >= 60 )
    System.out.println( "Passed " );
```

注意，Java 代码与伪代码非常相似，这是伪代码的一个特性，因此它是一种很有用的开发工具。if 结构体中的语句将字符串 “Passed” 输出到计算机屏幕上。

图 2.3 中所示的流程图解释了单选择 if 结构。这个流程图中包含了一个最重要的流程图符号，即菱形符号，也称为判断符。判断符包含一个可以为 true 或为 false 的表达式，比如一个条件。判断符发出两条流程线，一条表示判断符中表达式的值为 true，另一条表示判断符中表达式的值为 false。可以对任何结果为 boolean（布尔）类型值的表达式进行判断（例如，表达式的值为 true 或 false）。

注意，if 结构也是一个单入口 / 单出口结构。我们很快就会知道，其他控制结构的流程图（除了小圆圈和流程线之外）也只包含表示操作的矩形符号，以及表示判断的菱形符号。这就是我们所强调的程序设计的操作 / 判断模式。



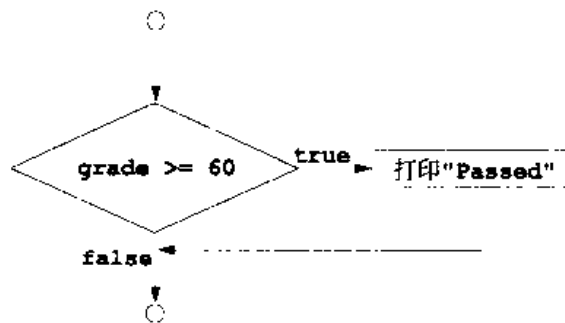


图 2.3 单选择 if 结构的流程图

我们可以设想有七个容器，每个容器中放入七种控制结构中的一种。这些控制结构都是空的，因此流程图中的矩形符号和菱形符号中都是空的。那么程序设计者的任务就是根据算法的需要使用各种控制结构来编写程序，按两种方式（堆栈或嵌套）将控制结构组合在一起，然后填入与算法对应的操作和判断。随后，我们将会讨论编写操作和判断的各种方法。

## 2.6 if / else 选择结构

只有在条件为 true 时，if 选择结构才会执行某个操作，否则就跳过该操作。if/else 选择结构则使程序员可以在条件为 true 或为 false 时分别指定执行不同的操作。例如下面的伪代码语句：

```
If    student's grade is greater than or equal to 60
    Print "Passed"
else
    Print "Failed"
```

如果学生的分数高于或等于 60 分，则打印 “Passed”；如果分数低于 60 分则打印 “Failed”。在打印之后，顺序 “执行” 下一条伪代码语句。注意，else 的程序体部分也是缩进的。

### 编程技巧 2.2

if/else 结构体中的语句都要缩进。

无论采用哪种缩进方式，都要在整个程序中小心使用。在一个程序中使用不同的缩进方式会使程序很难阅读。

将上述 if/else 结构的伪代码写成 Java 语句就是：

```
if ( grade >= 60 )
    System.out.println ( " Passed " );
else
    System.out.println ( " Failed" );
```

图 2.4 中的流程图清晰地展示了 if/else 结构的控制流程。注意，流程图中除了小圆圈和箭头符号以外，只有矩形（用于操作）和菱形（用于判断）两种符号。我们仍然要强调计算的这种操作/判断模式，假设有许多用于建立 Java 算法的空双选择结构，程序员的工作就是按照算法的需要将选择结构（使用嵌套或堆栈方式）与其他控制结构组合到一起，并根据算法的需要在空的矩形和菱形符号中填入相应的操作和判断。

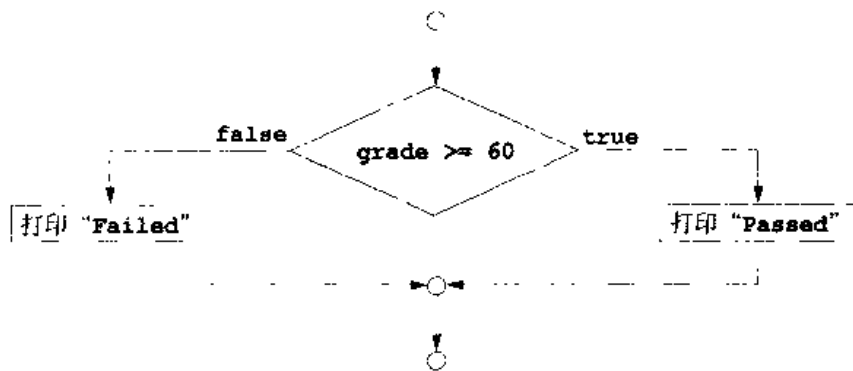


图 2.4 if/else 双选择结构的流程图

条件运算符 (?:) 是与 if/else 结构紧密相关的。“?:” 是 Java 中唯一的一个三元运算符，它需要三个操作数，与 “?:” 一起构成条件表达式。第一个操作数是一个布尔表达式，第二个操作数是当第一个布尔表达式的值为 true 时条件表达式的取值，第三个操作数是当布尔表达式的值为 false 时条件表达式的取值，例如，下列输出语句：

```
System.out.println( grade >= 60 ? " Passed " : " Failed " );
```

其功能是当条件 “grade>=60” 为 true 时打印字符串 “Passed”，当条件为 false 时打印字符串 “Failed”。因此，包含条件运算符的语句基本执行与前面的 if/else 语句一样的操作。由于条件运算符的优先级较低，因此一般将整个条件表达式放在括号中。条件运算符可以用在某些 if/else 语句不能使用的情况中。

### 编程技巧 2.3

一般情况下，条件表达式比 if/else 结构更难以阅读。当这种表达式有助于提高程序的可读性时，可以在判断中使用这样的表达式。

嵌套的 if/else 结构是将 if/else 结构放在 if/else 结构之中。例如，下面的伪代码将在考试成绩高于或等于 90 分时打印 A，考试成绩为 80 到 89 分时打印 B，70 到 79 分打印 C，60 到 69 分则打印 D，其他分数打印 F：

```

If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater or equal to 80
        Print "B"
    else
        If student's grade is greater or equal to 70
            Print "C"
        else
            If student's grade is greater or equal to 60
                Print "D"
            else
                Print "F"
  
```

该伪代码可写成如下所示的 Java 语句：

```

if ( grade >= 90 )
    System.out.println( "A" );
  
```

```
else
    if ( grade >= 80)
        System.out.println ( " B " );
    else
        if ( grade >= 70)
            System.out.println ( " C " );
        else
            if ( grade >= 60)
                System.out.println ( " D " );
            else
                System.out.println ( " F " );
```

如果 grade 大于或等于 90, 则前 4 个条件都为 true, 只执行第一个条件后的 System.out.println 语句。执行了该 System.out.println 语句后, 就跳过“外部”if/else 语句中的 else 部分。

#### 编程技巧 2.4

如果存在几层缩进, 那么每一层都应缩进相同的格数。

大多数 Java 程序员都习惯将上述 if 结构写成:

```
if (grade >= 90)
    System.out.println ( " A " );
else if ( grade >= 80 )
    System.out.println ( " B " );
else if ( grade >= 70 )
    System.out.println ( " C " );
else if ( grade >= 60 )
    System.out.println ( " D " );
else
    System.out.println ( " F " );
```

这两种格式的程序是一样的。由于后一种格式不必向右缩进很多, 因此这种格式的使用更为广泛。前一种缩进方式通常使一行语句向右缩进很多, 以至于有时不得不将语句折行, 这样就降低了程序的可读性。

需要注意的是, Java 编译器总是将 else 与离它最近的 if 组合在一起, 除非用花括号 ({} ) 指定不同的匹配方式。这称为“悬挂 else 问题”, 例如:

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println ( " x and y are > 5 " );
else
    System.out.println ( " x is <= 5 " );
```

从结构上来看, 该结构表示如果 x 大于 5, 而且 y 也大于 5, 则打印 “x and y are > 5”。否则, 如果 x 不大于 5, 则 if/else 结构中的 else 部分将输出 “x is <= 5”。

然而, 这个嵌套的 if 结构实际上并不像上面所述的那样去运行。编译器将上面的结构解释为:

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println ( " x and y are > 5 " );
    else
        System.out.println ( " x is <= 5 " );
```

其中，第一个 if 结构体中是一个 if/else 结构。这个结构判断 x 是否大于 5，如果 x > 5，则继续判断 y 是否也大于 5。如果第二个条件为 true，则正确地打印字符串 "x and y are > 5"。但是，如果第二个条件为 false，则将打印字符串 "x is <= 5"，尽管实际上 x 大于 5。

如果希望这个 if 嵌套结构按我们的设想去运行，则必须写为：

```
if ( x > 5 ) {  
    if ( y > 5 )  
        System.out.println( " x and y are > 5 " );  
}  
else  
    System.out.println( " x is <= 5 " );
```

花括号 ( {} ) 向编译器表明第二个 if 结构是第一个 if 结构的一部分，else 是与第一个 if 结构相对应的。在练习 2.21 和练习 2.22 中将更深入地研究悬挂 else 问题。

if 选择结构通常认为它的结构体中只有一条语句。如果要在 if 结构体中包含多条语句，可以使用花括号将这些语句括起来，括起来的多条语句称为复合语句。

#### 软件工程视点 2.2

复合语句可以放在任何单个语句可以放置的位置。

下面的例子中，在 if/else 结构的 else 部分包含了一条复合语句：

```
if ( grade >= 60 )  
    System.out.println( " Passed " );  
else {  
    System.out.println( " Failed " );  
    System.out.println( " You must take this course again. " );  
}
```

在这个例子中，如果 grade 少于 60，则程序将执行 else 体中的两条语句，并打印下面的信息：

```
Failed  
You must take this course again.
```

注意，else 中的两条语句是用花括号括起来的。如果没有花括号，那么下列语句：

```
System.out.println( " You must take this course again. " );
```

就不属于 if 结构的 else 部分，不论 grade 是否少于 60，这条语句都将执行。

#### 常见编程错误 2.2

如果丢失了复合语句的一个或一对括号，则将导致语法错误或逻辑错误。

语法错误（比如复合语句中缺少一个花括号）可以由编译器发现。而逻辑错误（比如复合语句中缺少两个花括号）则会影响运行的结果，致命的逻辑错误将导致程序过早地终止或运行失败。非致命的逻辑错误虽然不会终止程序运行，但会产生不正确的结果。

#### 软件工程视点 2.3

复合语句可以像单个语句一样放在程序中的任何位置；同样，程序中也可以没有任何语句（即使用空语句）。空语句只包括一个分号 (;)，可以放在任何位置。

### 常见编程错误 2.3

在if结构中，将分号放在条件语句之后会使单选择if结构产生逻辑错误，使双选择if结构（如果if部分确实有语句）产生语法错误。

### 编程技巧 2.5

有些程序员喜欢在输入语句之前先输入开始和结束的花括号。这样可以避免漏掉花括号。

在本节中，我们介绍了复合语句的概念。复合语句可以包含声明（与main程序类似），如果包含了声明，那么复合语句也称为“块”。块中的声明一般都位于块的开始处，即在操作语句的前面。但声明也可以与操作语句混合放在一起。

## 2.7 while 循环结构

程序员可以使用循环结构来控制在某些条件为true时重复执行一个操作。下列伪代码语句：

```
While there are more items on my shopping list  
    Purchase next item and cross it off my list
```

描述了购物过程中的循环动作。条件“there are more items on my shopping list”可以为true也可以为false。如果为true，则执行操作“Purchase next item and cross it off my list”。如果条件始终为true，则重复执行该操作。while循环结构中的语句构成了while结构体，while结构体既可以是单个语句，也可以是复合语句。循环条件最终会变为false（当购物单中的最后一项已经买完，并从购物单中划掉时，该条件就会变为false，此时循环将会终止，并继续执行循环结构后的语句。

### 常见编程错误 2.4

如果在while结构体中未能提供使while的循环条件最终变为false的操作，则会产生错误。这样的循环结构将永远不会终止，称为“无限循环”错误。

### 常见编程错误 2.5

如果将关键字while的第一个字母写成大写的W（While），则会产生错误（记住，Java是区分大小写的语言）。Java的所有关键字（比如while、if和else）都只包含小写字母。

下面分析一个while结构的例子，找出2的幂次运算结果中大于1000的第一个值。假设将变量product初始化为2，当下面的while结构执行完毕时，product中保存的值即为所求的结果：

```
int product = 2;  
while (product <= 1000 )  
    product = 2 * product;
```

图2.5的流程图说明了上面的while循环结构的控制流程。注意，在这个流程图中，除了小圆圈和箭头符号，只包含一个矩形符号和一个菱形符号。我们可以设计一个很大的空while结构，它可以和其他的控制结构组合在一起，以实现某个算法的控制流程，然后再向这个空while结构的矩形和菱形符号中填入适当的动作和判断条件。这个流程图清晰地表示了上述循环结构，从矩形上画出的流程线又返回到判断条件上，每循环一次就对该条件进行一次判断，直到该条件最终变为false。此时，将退出while循环，并继续执行程序的下一条语句。

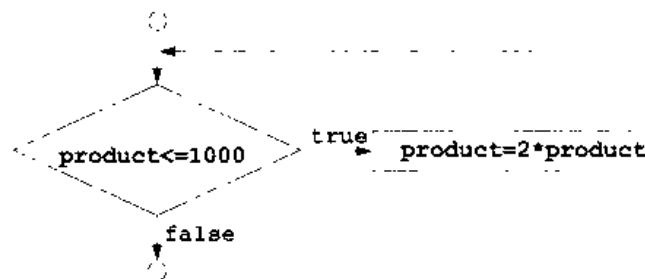


图 2.5 while 循环结构的流程图

在进入 while 结构时, 变量 `product` 的值为 2。然后 `product` 反复地乘 2, 其值分别为 4、8、16、32、64、128、256、512, 最后为 1024。当 `product` 的值变为 1024 时, while 结构的判断条件 “`product <= 1000`” 就变为 `false`, 这时循环将终止; 然后, 继续执行 while 后的下一条语句。注意: 如果一个 while 结构的判断条件的初始值为 `false`, 那么其结构体中的语句将永远不会执行。

## 2.8 构造算法: 实例 1 (计数器控制循环)

为了说明算法是如何开发的, 我们来看一个计算班级平均成绩的实际问题。考虑下面的问题:

一个班级有 10 个学生参加了一次测验。根据每个人的测验成绩 (用字母 A、B、C、D 或 F 表示), 求出全班的平均成绩。

通过将字母表示的成绩换算成分数值来计算平均值, 成绩 A 的分数值为 4, B 的值为 3, C 的值为 2, D 的值为 1, F 的值为 0, 班级平均成绩等于所有学生分数值的总和除以学生总数。在计算机上解决该问题时必须先输入每个人的成绩, 然后计算平均值, 最后打印结果。

我们先用伪代码列出要执行的动作, 然后再确定这些动作的执行顺序。我们使用计数器控制循环来输入学生成绩, 一次输入一个值。这种循环采用一个计数器变量, 指定语句集的执行次数。在这个例子中, 当计数器中的计数超过 10 时, 将终止循环。这里我们分别给出了伪代码算法 (如图 2.6 所示) 和相应的程序 (如图 2.7 所示)。在下一节中, 我们将介绍这个伪代码算法是如何开发的。计数器控制循环通常称为确定循环, 因为在循环开始执行之前, 就已经知道循环的次数了。

注意, 算法中有两个变量: `total` 和 `counter`。变量 `total` 用于累加所有学生的分数值。变量 `counter` 用于计数, 在本例中, 它用于计算已输入成绩的个数。在使用变量 `total` 之前, 通常都将其初始化为 0; 否则, 最后得到的总和就会包括原来保存在 `total` 中的值。作为计数器的变量通常都初始化为 0 或 1, 这要根据不同的使用方式而定 (后面将提供这样的例子)。

在图 2.7 的程序中, Java 编译器将确保变量 `counter` 和 `total` 在用子计算之前都进行了初始化。如果计算时使用了未初始化的变量, 则编译器会显示出错误信息, 告知用户这些变量可能未初始化。

### 编程技巧 2.6

应初始化各个计数器变量和累加值变量。

```
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
Input the next grade
```

```

    If the letter grade is equal to A
        Add grade point value 4 to the total
    else if the letter grade is equal to B
        Add grade point value 3 to the total
    else if the letter grade is equal to C
        Add grade point value 2 to the total
    else if the letter grade is equal to D
        Add grade point value 1 to the total
    else if the letter grade is equal to F
        Add grade point value 0 to the total

    Add one to the grade counter
    Set the class average to the total divided by ten
    Print the class average

```

图 2.6 使用计数器控制循环来解决班级平均成绩问题的伪代码算法

```

1  // Fig. 2.7: Average.java
2  // Class average program with
3  // counter-controlled repetition
4  import java.io.*;
5
6  public class Average {
7      public static void main( String args[] ) throws IOException
8      {
9          int counter, grade, total, average;
10
11         // initialization phase
12         total = 0;
13         counter = 1;
14
15         // processing phase
16         while ( counter <= 10 ) {
17             System.out.print( "Enter letter grade: " );
18             System.out.flush();
19             grade = System.in.read();
20
21             if ( grade == 'A' )
22                 total = total + 4;
23             else if ( grade == 'B' )
24                 total = total + 3;
25             else if ( grade == 'C' )
26                 total = total + 2;
27             else if ( grade == 'D' )
28                 total = total + 1;
29             else if ( grade == 'F' )
30                 total = total + 0;
31
32             System.in.skip( 1 );    // skip the newline character
33             counter = counter + 1;
34         }
35
36         // termination phase
37         average = total / 10;      // integer division
38         System.out.println( "Class average is " + average );

```

```
39     }  
40 }
```

**输出结果：**

```
Enter letter grade:A  
Enter letter grade:A  
Enter letter grade:A  
Enter letter grade:A  
Enter letter grade:A  
Enter letter grade:B  
Enter letter grade:B  
Enter letter grade:C  
Enter letter grade:C  
Enter letter grade:F  
Class average is 2
```

图 2.7 采用计数器控制循环来解决班级平均成绩问题的 Java 程序

这是我们编写的第一个 Java 应用程序，因此在继续讲解之前，有些特别的属性必须先解释一下。一个应用程序从 main 方法开始执行（第 7 行）。应使用 Java 解释器（java）来执行应用程序，而不是使用 appletviewer 或 WWW 浏览器。

第 4 行的语句：

```
import java.io.*;
```

用于引入 Java 的输入/输出软件包，使我们的程序可以从键盘上读入数据，也可以向屏幕上输出数据。

在创建应用程序时，必须定义一个类，就像我们在第一章中所做的。不过，这个类不是 Applet 的扩展类，因为我们并不是创建一个 applet。在这个应用程序中，在 Average 类中定义 main 方法。main 方法的第一条语句必须如下所示：

```
public static void main( String args [ ] )
```

如果 main 方法不是以该语句开头，那么 Java 解释器将无法执行这个程序。在后面的章节中，我们将解释为什么这条语句如此重要。现在，如果要创建一个应用程序，只需直接把该语句添加到 main 方法中即可。

在图 2.7 中，main 方法实际上是以下面的语句开头的：

```
public static void main( String args [ ] )throws IOException
```

这个语句包含了一些可选信息。用户将从键盘上输入学生的成绩（用字母表示），这些数据用于计算班级平均成绩。当从键盘上读入信息时，Java 将测试输入过程中是否发生了问题（比如，不再有数据输入）。如果发生了问题，那么 Java 就会产生一个异常，告诉程序员产生了什么错误。顾名思义，“异常”就是指程序在执行过程中出现的意外情况，它们不是经常发生的，但是有可能会发生。程序员可以处理这些异常，使程序能够继续执行；也可以忽略这些异常，这时程序将终止。

Java 对于异常处理特别严格。如果编译一个包含键盘输入的程序，那么 Java 会“期望”程序中能够处理“输入异常”。若想忽略这些异常，就必须向编译器指明已经意识到了这些可能发生的问题。第 7 行结尾的“throws IOException”就是向编译器指出：我们已意识到当程序试图从键盘读入数据时可能会产生输入/输出异常，但在程序中我们将有意忽略这些异常，这称之为声明异常。“throws IOException”称为 main 方法的 throws 子句。



我们在这里选择忽略异常是为了可以着重讨论算法和控制结构,本书后面的内容中将详细介绍对各种异常情况的处理。

如果我们没有声明异常,编译程序就会给出下面的错误信息:“Exception java.IOException must be caught, or it must be declared in the throws clause of this method.”(必须处理java.io.IOException异常,否则就必须在该方法的 throws 子句中声明该异常。)

下面的各语句对应于伪代码语句“Input the next grade”:

```
System.out.print( " Enter letter grade : " );
System.out.flush( );
grade = System.in.read( );
```

前两条语句用于在屏幕上显示下列提示:“Enter letter grade:”。Java 中所有向屏幕上的输出动作和来自键盘的输入动作都由“流”来完成。因此,第一行的语句就是向标准的输出流对象 System.out (该对象通常都是与屏幕“相联系的”)发送字符流“Enter letter grade:”。System.out.print 方法带有一个参数,其功能是将该参数输出以显示在屏幕上。使用该方法时,有时会将要输出的字符串放到一个缓冲区中(计算机内存中的一个临时存放区),而并不是显示到屏幕上。System.out.flush 方法将在用户输入成绩之前强制显示缓冲区中的字符串。

前面的第三行语句是使用标准的输入流对象 System.in 来获取用户输入的成绩值。System.in 对象用于获取来自标准输入流(通常是键盘)的输入。System.in.read 方法从键盘上读入一个字符,并将该字符保存在整数变量 grade 中。字符既可以保存在两字节的 char 类型变量中,也可以保存在四个字节的 int 类型变量中。这样,我们就可以根据不同的用处将一个字符看做是整数或字符类型。

例如,下面的语句:

```
System.out.print( " The character ( " + 'a' + " ) has the value" + ((int)'a') );
```

将输出字符 a 及其相应 ASCII 代码的整数值,如下所示:

```
The character (a) has the value 97
```

整数 97 是该字符在计算机内的数字表示。Java 采用 ISO (国际标准化组织, International Standards Organization) 的 Unicode 字符集。在该字符集中,97 表示小写字母 a。现在,很多计算机都使用 ASCII (美国信息交换标准码, American Standard Code for Information Interchange) 字符集,ASCII 字符集是 Unicode 字符集的一个子集,本书的附录中列出了 ASCII 字符及其对应的数值。

请注意前面语句中字符 a 两边所用的单引号。当程序中包含一个字符的字母值时,该字符必须放在一对单引号 (') 中。如果将它放在双引号 (") 中,Java 就会把该字符解释为一个字符串。字符串不能直接赋给 char 类型或 int 类型的变量。

当用户输入了一个成绩之后,程序将在一个 if/else 结构中处理该成绩。注意,每条 if 语句的条件都是判断变量 grade 的值是否等于一个字符值,该字符放在一对单引号中,表示相应的成绩。

接着,程序将计数器变量加 1,表明已处理完一个成绩。不过,当用户从键盘上输入一个字符并按下回车键时,实际上将有两个字符通过输入流输送到程序中:一个是字母成绩(比如“A”),另一个是换行符。当用户按下回车键时,换行符会自动输入到程序中(这也会导致换行符输出到屏幕上,使得下一个输出字符出现在屏幕下一行的开始处)。下面的语句表示程序应在输入流中跳过 1 个字符(换行符):

```
System.in.skip ( 1 ); // skip the newline character
```

这样就使用户在程序返回第 19 行准备读入下一个成绩时,能够从键盘上输入一个成绩。如果没有这条语句,程序就会将换行符作为一个合法的字母成绩来处理。最后,使用下面的语句来显示字符串“Class average is”及变量 average 的值:

```
System.out.println ( " Class average is" + average );
```

System.out.println 方法的功能是输出其参数,后跟一个换行符。这条语句后不需要再使用 System.out.flush 方法,因为 System.out.println 将自动强制输出显示在屏幕上。

如果要执行该程序(已编译完之后),只需在计算机的命令行上(在 Windows 95 或 Windows NT 系统的 DOS 提示符下,或者在 UNIX 系统的 shell 提示符下)输入下面的命令,然后按下回车键:

```
java Average
```

这将执行 Java 解释器,并告诉它启动该应用程序在 Average 类中的 main 方法。如果对于从计算机的命令行上执行命令有问题,可以请教指导老师。

注意,该程序中的平均值计算将产生一个整数结果。实际上,本例中的分数值总和是 29,当该值除以 10 时就等于 2.9,即一个带有小数点的数字。在下一节中,我们将会看到如何处理这种数字(称为浮点数)。

## 2.9 自顶向下、逐步求精的构造算法:实例 2(标志控制循环)

让我们扩展一下班级平均成绩的问题。考虑下面的问题:

开发一个计算班级平均成绩的程序,每次运行该程序时可以处理任意多个成绩。

在第一个计算班级平均成绩的例子中,预先已知道要处理的成绩数(10)。在这个例子中,没有指明要输入多少个成绩,该程序必须能处理任意个数的成绩。那么在程序中如何确定何时停止输入成绩呢?程序怎样知道何时计算和打印班级平均成绩呢?

解决这个问题的一個方法是使用一个称为标志值的特殊值(也称为信号值、哑值)作为“数据输入结束”的标志。用户输入所有合法的成绩后,输入标志值,表示最后一个成绩已经输入。标志控制循环通常也称为不确定循环,因为在循环开始执行之前并不知道循环的次数。

显然,标志值必须经过挑选,使它不会与合法的输入值相混淆。因为测验的成绩包括字母 A、B、C、D 和 F,所以可以把字母 Z 作为标志值。这样,运行上述的班级平均成绩的程序时,就可以处理以下的输入流: A、A、C、C、B 和 Z。然后再根据成绩 A、A、C、C 和 B,计算并打印班级平均成绩(Z 是标志值,因此它不应参加平均值的计算)。

### 常见编程错误 2.6

选择一个合法的数据值作为标志值会导致逻辑错误,而且可能会使标志控制循环无法正确终止。

我们采用一种称为自顶向下、逐步求精的方法来解决班级平均成绩的问题,这种方法对于开发一个具有良好结构的算法是必不可少的。我们从用伪代码描述的“顶部”开始:

```
Determine the class average for the quiz
```

“顶部”是一个句子,表达程序的总体功能。实际上,“顶部”应该是对一个程序的完整描述。但是,“顶部”很少能将编写 Java 算法的细节充分表达出来,因此应进一步求精。将“顶部”划分为一系列更小的任务,并按照所需的处理顺序把它们列出来。这样就得到了下面的第一步求精结果:

```
Initialize variables
Input, sum and count the quiz grades
Calculate and print the class average
```

在这里只用到了顺序结构,所列出的各个步骤将按顺序一个接一个地执行。

#### 软件工程视点 2.4

每一步求精的结果,再加上“顶部”本身,就是一个完整的算法说明,只有细节的层次是可变的。

为了继续进行下一步求精(即第二步求精),需要处理具体的变量。这里需要一个累加成绩的变量total,一个统计已处理个数的计数器counter,一个用于接收用户输入的成绩值的变量,以及一个存放计算出来的平均值的变量。下面的伪代码语句:

```
Initialize variables
```

可以细化为:

```
Initialize total to zero
Initialize counter to zero
```

只有变量total和counter需要在使用之前初始化;变量average和grade(分别用于存放平均值和用户输入的成绩)不必进行初始化,因为它们的值在进行计算或输入时重写。

对于下面的伪代码语句:

```
Input ,sum , and count the quiz grades
```

需要使用一个循环结构(即一个while循环)才能成功地输入每个成绩。因为我们事先并不知道要处理多少个成绩,因此应使用标志控制循环。用户将一次一个地输入合法的成绩。当最后一个合法的成绩输入完后,用户就会输入标志值,程序将在每次输入值后判断当前的值是否是标志值,如果用户输入的是标志值,就终止循环。这样,上述伪代码语句经过第二步求精就变为:

```
Input the first grade (possibly the sentinel )
While the user has not as yet entered the sentinel value
    If the letter grade is equal to A
        Add grade point value 4 to the total
    else if the letter grade is equal to B
        Add grade point value 3 to the total
    else if the letter grade is equal to C
        Add grade point value 2 to the total
    else if the letter grade is equal to D
        Add grade point value 1 to the total
    else if the letter grade is equal to F
        Add grade point value 0 to the total

    Add one to the grade counter
    Input the next grade (possibly the sentinel )
```

注意,在伪代码中我们没有使用花括号将构成while结构体的语句括起来,只是简单地将这些语句在while下面进行了缩进,表示它们都属于while结构。毕竟,伪代码只是一种非正式的程序开发辅助手段。

下列伪代码语句:

```
Calculate and print the class average
```

可以细化为以下各步:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print " No grades were entered "
```

注意, 我们在这里小心地判断了是否可能出现除数为零的情况, 这将是一个逻辑错误, 如果没有检测出来就会导致程序产生不正确的输出。图 2.8 给出了求解班级平均成绩问题的完整的第二步求精伪代码。

```
-----
Initialize total to zero
Initialize counter to zero

input the first grade (possibly the sentinel )
While the user has not as yet entered the sentinel value
    If the letter grade is equal to A
        Add grade point value 4 to the total
    else if the letter grade is equal to B
        Add grade point value 3 to the total
    else if the letter grade is equal to C
        Add grade point value 2 to the total
    else if the letter grade is equal to D
        Add grade point value 1 to the total
    else if the letter grade is equal to F
        Add grade point value 0 to the total
        Add one to the grade counter
    Input the next grade (possibly the sentinel )

    If the counter is not equal to zero
        Set the average to the total divided by the counter
        Print the average
    else

        Print "No grades were entered"
-----
```

图 2.8 使用标志控制循环来解决班级平均成绩问题的伪代码算法

#### 测试与调试提示 2.1

当执行一个除数表达式可能为零的除法时, 应显式地判断这种情况, 并在程序中进行适当的处理 (比如打印一个出错信息), 而不要任由除数为零的情况发生。

#### 编程技巧 2.7

在伪代码程序中包含一些空行, 可以使伪代码具有更强的可读性。空行用于分隔伪代码的控制结构, 也就是给程序分段。

#### 软件工程视点 2.5

很多算法在逻辑上都可以分成三个段: 一个初始化段, 用于初始化程序中的变量; 一个处理段, 用于输入数据值, 并相应地调整程序的变量; 一个终止段, 用于计算和打印最终结果。

图2.8中所示的伪代码算法解决了更一般的班级平均成绩问题。本算法只经过了两层求精，有些算法可能需要更多层的求精。

#### 软件工程视点 2.6

当伪代码算法已经足够细化，程序员能够将该伪代码转换成一个Java应用程序或applet时，程序员就可以终止自顶向下、逐步求精的过程。随后的工作通常是实现该Java算法。

图2.9中给出了该Java程序及一次执行示例。尽管每个成绩的分数值都是一个整数，但平均值的计算结果仍可能产生一个带小数点的数字（即一个实数）。类型int无法表示实数，因此该程序中引入了数据类型double，用于处理带小数点的数字（也称为浮点数）；还引入了一个特殊的运算符，称为强制类型转换（cast）运算符，用于处理计算平均值时的类型转换问题。这些特性在后面的内容中将会详细解释。

---

```
1      // Fig. 2.9: Average.java
2      // Class average program with
3      // sentinel-controlled repetition.
4      import java.io.*;
5
6      public class Average {
7          public static void main( String args[] ) throws IOException
8          {
9              double average; // number with decimal point
10             int counter, grade, total;
11
12             // initialization phase
13             total = 0;
14             counter = 0;
15
16             // processing phase
17             System.out.print( "Enter letter grade, Z to end: " );
18             System.out.flush();
19             grade = System.in.read();
20
21             while ( grade != 'Z' ) {
22                 if ( grade == 'A' )
23                     total = total + 4;
24                 else if ( grade == 'B' )
25                     total = total + 3;
26                 else if ( grade == 'C' )
27                     total = total + 2;
28                 else if ( grade == 'D' )
29                     total = total + 1;
30                 else if ( grade == 'F' )
31                     total = total + 0;
32
33                 System.in.skip( 1 );
34                 counter = counter + 1;
35                 System.out.print( "Enter letter grade, Z to end: " );
36                 System.out.flush();
37                 grade = System.in.read();
38             }
39
40             // termination phase
```

---

```

41         if ( counter != 0 ) {
42             average = (double) total / counter;
43             System.out.println( "Class average is " + average );
44         }
45         else
46             System.out.println( "No grades were entered" );
47     }
48 }

```

**输出结果:**

```

Enter letter grade,z to end : A
Enter letter grade,z to end : A
Enter letter grade,z to end : A
Enter letter grade,z to end : A
Enter letter grade,z to end : A
Enter letter grade,z to end : B
Enter letter grade,z to end : B
Enter letter grade,z to end : B
Enter letter grade,z to end : B
Enter letter grade,z to end : B
Enter letter grade,z to end : Z
Class average is 3.5

```

图 2.9 使用标志控制循环来处理班级平均成绩问题

在这个例子中,控制结构可以一个接一个地堆放在一起(按顺序),就像小孩堆积木一样,while 结构后面紧跟着一个 if/else 结构。本例中还使用了 Java 的另一种控制结构的组织方式,即一个控制结构可以嵌套在另一个控制结构中,比如在 while 结构中嵌套了一个 if/else 结构。

注意图 2.9 的 while 循环中的复合语句。如果没有花括号,循环体中最后三个语句就会落在循环体的外面,则这段代码将会错误地解释成下面的形式:

```

while( grade!= ' Z ' )
    if ( grade== 'A' )
        total = total +4 ;
    else if ( grade == ' B ' )
        total = total +3 ;
    else if( grade == ' C ' )
        total = total + 2 ;
    else if( grade == ' D ' )
        total = total + 1 ;
    else if( grade == ' F ' )
        total = total + 0 ;

System.in.skip ( 1 );
counter = counter + 1;
System.out.print ( " Enter letter grade , z to end : " );
System.out.flush ( );
grade = System.in.read ( );

```

如果用户输入的第一个成绩不是“Z”,那么这段程序会导致一个无限循环。

**编程技巧 2.8**

在一个标志控制循环中,应该使用一个数据输入提示语句来显式地提醒用户标志值是什么。

平均值的结果并不总是整数值。通常，平均值总是像3.333或2.7这样的带有小数部分的值。这些值称为浮点数，使用数据类型 `double` 来表示。将变量 `average` 声明为 `double` 类型，以便保存计算所得的小数结果。不过，表达式“`total/counter`”的计算结果是一个整数，因为 `total` 和 `counter` 都是整数变量。两个整数相除的结果仍然是整数，即丢弃商的小数部分（截尾）。因为该计算是首先完成的，所以在将计算结果赋给 `average` 之前，其小数部分就已经丢弃。为了使用整数值进行浮点计算，必须创建临时的用于计算的浮点值。Java提供了一元的强制类型转换操作来完成这个任务。下面的语句中包含了一个强制类型转换运算符“`(double)`”，用于将其操作数 `total` 转换成一个临时的浮点数：

```
average = ( double ) total/counter;
```

这种使用强制类型转换运算符的方式称为显式转换。保存在 `total` 中的值仍然是整数。现在，该计算就变成使用一个浮点数（`total` 的临时 `double` 值）除以一个整数 `counter`。Java编译器只知道如何计算操作数类型一致的表达式。为确保操作数具有相同的类型，编译器将对指定的操作数执行“提升”操作（也称为隐式转换）。例如，在一个同时包含 `int` 和 `double` 类型的表达式中，`int` 类型的操作数会“提升”为 `double` 类型。在上例中，`counter` 首先提升为 `double` 类型，然后再完成计算，最后将浮点类型的结果赋给变量 `average`。本章后面将介绍所有的标准数据类型及其提升顺序。

#### 常见编程错误 2.7

强制转换运算符可以用于实现基本数据类型之间及相关类型之间的转换。将变量强制转换为错误的类型将导致编译错误或运行时错误。

任何数据类型都可以使用强制类型转换。强制类型转换运算符由一个数据类型名字及其外面的括号组成。强制类型转换运算符是一个一元运算符，即只带有一个操作数的运算符。在第1章中，我们曾学习了二元算术运算符。Java还支持一元的正（`+`）和负（`-`）运算符，因此，程序员可以使用诸如 `-7` 或 `+5` 这样的表达式。强制类型转换运算符按从左向右的顺序结合，有与其他一元运算符（比如一元“`+`”和一元“`-`”）相同的计算优先级；其优先级比运算符 `*`、`/` 和 `%` 要高，但比括号低。在优先级表中，我们用标记“`(type)`”表示强制类型转换运算符。

#### 常见编程错误 2.8

将浮点数作为精确值来使用会导致错误的结果，大多数计算机都只能近似地表示浮点数。

#### 常见编程错误 2.9

假设整除就是取整（而不是截尾）将得出错误的结果。

#### 编程技巧 2.9

不要用等于或不等于来比较浮点值，而应该判断它们之间的差是否小于某一指定值。

尽管浮点数并不总是“百分之百的准确”，但是我们仍在大量地应用浮点数。例如，当我们表述正常的体温是98.6 K时，并不需要精确到很小的值。在查看体温表时，若该数为98.6，那么它实际上可能是98.599 947 321 064 3。在大多数情况下，最好将该值简单地读为98.6。

浮点数的另一个用处是进行除法。当用10除以3时，结果是3.333 333...，小数点后是无限循环的3。计算机只能为该值分配一个固定大小的空间，因此所保存的值只能是一个近似值。

## 2.10 自顶向下、逐步求精的构造算法——实例3（嵌套的控制结构）

让我们开始解决另一个实际问题。这里将再一次使用伪代码来描述算法，然后自顶向下、逐步

求精，最后编写出相应的 Java 程序。考虑下面的问题：

一个大学开设了一门课程，帮助学生参加不动产经纪人的国家资格考试。去年，有一些学生选修了这门课，并参加了资格考试。现在，学校想要了解学生的考试情况，请你编写一个程序，用来统计测验结果。现有一份 10 名学生的名单，每个学生的名字后面有一个数字，如果通过了考试则为 1，如果未通过考试则为 2。

在程序中应按如下步骤分析测验结果：

1. 输入每个测验结果（即 1 或 2）。每当程序需要下一个测验结果时，就在屏幕上显示信息“Enter result”。
2. 计算两种测验结果的数目。
3. 显示测验结果的一个总结，指出通过考试的学生数和未通过考试的学生数。
4. 如果通过考试的学生多于 8 个，就打印信息“Raise tuition”。

仔细地阅读前面的问题之后，我们得出如下结论：

1. 该程序必须处理 10 个学生的测验结果，应使用一个计数器控制循环。
2. 每个测验结果都是一个数字——1 或 2。每当程序读入一个测验结果时，都必须确定该数字是 1 还是 2。在我们的算法中，只判断是否为 1。如果该数字不是 1，就假定它是 2（在本章后面的一个练习中考虑了这一假设的推论）。
3. 使用两个计数器来记录测验结果，一个用于计算通过考试的学生数目，一个用于计算未通过考试的学生数目。
4. 当程序处理完所有的结果之后，必须判断是否有 8 个以上的学生通过了考试。

让我们采用自顶向下、逐步求精的方法来处理该问题。首先给出用伪代码表示的“顶部”：

```
Analyze exam results and decide if tuition should be raised
```

强调“顶部”是程序的一个完整描述仍然很重要，但在将伪代码直接写成 Java 程序之前，可能需要几步求精的过程。第一步求精结果是：

```
Initialize variables  
Input the ten exam grades and count passes and failures  
Print a summary of the exam results and decide if tuition should be raised
```

尽管这已经是对整个程序的完整描述了，但仍然需要更进一步的求精。现在先处理特定的变量。需要两个计数器来记录通过的人数和未通过的人数，还要一个控制循环过程的计数器，以及一个用来保存用户输入的变量。伪代码语句：

```
Initialize variables
```

可以求精为：

```
Initialize passes to zero  
Initialize failures to zero  
Initialize student to one
```

注意，只有记录通过人数的计数器、未通过人数的计数器和学生总数的计数器进行了初始化。下一条伪代码语句：



---

```
Input the ten quiz grades and count passes and failures
```

需要使用一个循环才能成功地输入每个人的测验结果。这里事先已精确地知道了有10个测验结果, 因此应使用计数器控制循环。在循环内部 (即在循环中嵌套), 将使用一个双重选择结构来判断每个测验结果是通过还是未通过, 并将相应的计数器递增。这样, 上面的伪代码语句就求精为:

```
while student counter is less than or equal to ten
    Input the next exam result
    If the student passed
        Add one to passes
    else
        Add one to failures
    Add one to student counter
```

注意, 语句中所用的空行将 if/else 控制结构分离出来, 从而增加了程序的可读性。下一条伪代码语句:

```
Print a summary of the exam results and decide if tuition should be raised
```

可以求精为:

```
Print the number of passes
Print the number of failures
If more than eight students passed
Print "Raise tuition"
```

图 2.10 中给出了第二步求精的完整语句。注意, 语句中也使用了空行将 while 结构分离出来, 以增加程序的可读性。

---

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passed
    else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed
Print "Raise tuition"
```

---

图 2.10 求解测验结果问题的伪代码

现在, 我们已经充分细化了上述伪代码, 可以将其转换成 Java 程序, 如图 2.11 所示。

---

```
1 // Fig. 2.11: Analysis.java
2 // Analysis of examination results
3 import java.io.*;
```

```

4
5     public class Analysis {
6         public static void main( String args[] ) throws IOException
7         {
8             // initializing variables in declarations
9             int passes = 0, failures = 0, student = 1,result;
10
11             // process 10 students; counter-controlled loop
12             while ( student <= 10 ) {
13                 System.out.print( "Enter result (1=pass,2=fail): " );
14                 System.out.flush();
15                 result = System.in.read();
16
17                 if ( result == '1' )          // if/else nested in while
18                     passes = passes + 1;
19                 else
20                     failures = failures + 1;
21
22                 student = student + 1;
23                 System.in.skip( 1 );
24             }
25
26             System.out.println( "Passed" + passes );
27             System.out.println( "Failed" + failures ) ;
28
29             if ( passes > 8 )
30                 System.out.println( "Raise tuition " );
31         }
32     }

```

**输出结果:**

```

Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):2
Enter result (1=pass,2=fail):2
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):2
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):2
passed 6
Failed 4

```

```

Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):2
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
Enter result (1=pass,2=fail):1
passed 9
Failed 1
Raise tuition

```

图 2.11 求解测验结果问题的 Java 程序及运行示例

注意，我们已充分利用了 Java 的一个特性，即允许在声明中进行变量的初始化。循环程序可能需要每次循环的开始对变量进行初始化，这样的初始化通常使用赋值语句来完成。

#### 编程技巧 2.10

在方法中声明的变量都应初始化，这样可避免编译器显示“未初始化的数据”的警告信息。基本数据类型 `int` 的实例变量将自动初始化为 0，基本数据类型 `boolean` 的实例变量将自动初始化为 `false`。

#### 软件工程视点 2.7

经验表明，在计算机上解决一个问题时的最困难部分就是开发相应的算法。一旦确定了正确的算法，从该算法产生相应的 Java 程序就很容易了。

#### 软件工程视点 2.8

很多有经验的程序员在编写程序时都不使用像伪代码这样的程序开发工具。这些程序员觉得他们的最终目标是在计算机上解决问题，而编写伪代码只会耽误时间。尽管对于简单熟悉的问题可以这样做（不编写伪代码），但对于大型、复杂的项目，这样做就可能会导致严重的错误。

## 2.11 赋值运算符

Java 提供了几种赋值运算符来简写赋值表达式。例如，下列语句：

```
C = C + 3;
```

可以使用加赋值运算符“`+=`”简写为：

```
C += 3;
```

“`+=`”运算符的功能是将运算符右边表达式的值与左边变量的值相加，然后将相加的结果赋给运算符左边的变量。任何具有如下形式的语句：

```
variable=variable operator expression;
```

都可以写成以下形式，其中 `operator` 是 `+`、`-`、`*`、`/` 或 `%`（以及其他二元运算符）中的一个：

```
variable operator =expression;
```

这样，“`C += 3`”就是将 `C` 的值加上 3。图 2.12 中给出了各种算术赋值运算符，以及使用这些操作的示例表达式和相应的解释。

赋值运算符	示例	解释	赋值
假设：int c = 3, d = 5, e = 4, f = 6, g = 12;			
<code>+=</code>	<code>c+=7</code>	<code>c=c+7</code>	c 为 10
<code>-=</code>	<code>d-=4</code>	<code>d=d-4</code>	d 为 1
<code>*=</code>	<code>e*=5</code>	<code>e=e*5</code>	e 为 20
<code>/=</code>	<code>f/=3</code>	<code>f=f/3</code>	f 为 2
<code>%=</code>	<code>g%=9</code>	<code>g=g%9</code>	g 为 3

图 2.12 算术赋值运算符

#### 性能提示 2.1

当使用“简写的”赋值运算符时，程序员在编写程序时可以更有效率，编译器编译程序也可以更快。有些编译器在使用“简写的”赋值运算符时，甚至可以使生成的代码运行得更快。

### 性能提示 2.2

我们在这一节所提到的很多性能提示都不会使性能有明显的提高,因此读者可以暂时忽略它们,只有在—一个循环很多次的循环程序中采用这种改善性能的技巧,才有可能得到显著的性能改善。

## 2.12 自增和自减运算符

Java 提供了一元自增运算符“++”和一元自减运算符“--”,在图 2.13 中列出了这些运算符。如果变量 C 加 1,那么可以使用自增运算符的表达式“C++”来代替表达式“C = C+1”或“C += 1”。如果自增运算符或自减运算符放在变量的前面,那么它们分别称为“前置自增运算符”或“前置自减运算符”。如果自增运算符或自减运算符放在变量的后面,那么它们分别称为“后置自增运算符”或“后自减运算符”。对一个变量进行前置自增(前置自减)运算,表示先将该变量的值加 1(减 1),然后在出现该变量的表达式中使用它的新值进行计算。对一个变量进行后置自增(后置自减)运算,表示在出现该变量的表达式中先使用它的原值进行计算,然后再将该变量加 1(减 1)。

运算符	名称	示例	解释
++	前置自增	++a	a 先加 1,然后在出现 a 的表达式中使用 a 的新值计算
++	后置自增	a++	先用 a 的原值计算,然后 a 再加 1
--	前置自减	--b	b 先减 1,然后在出现 b 的表达式中使用 b 的新值计算
--	后置自减	b--	先用 b 的原值计算,然后 b 再减 1

图 2.13 自增和自减运算符

图 2.14 的 applet 说明了“++”自增运算符执行前置自增和后置自增计算之间的区别。在第 13 行中,对变量 c 执行的是后自增运算,使得 c 在执行了调用 g.drawString 方法的操作之后才加 1。在第 19 行中,对变量 c 执行的是前置自增运算,使得 c 在执行调用 g.drawString 方法的操作之前就加 1。

```

1      // Fig. 2.14: Increment.java
2      // Preincrementing and postincrementing
3      import java.awt.Graphics;
4      import java.applet.Applet;
5
6      public class Increment extends Applet {
7          public void paint( Graphics g )
8          {
9              int c;
10
11              c = 5;
12              g.drawString( Integer.toString( c ), 25, 25 );
13              g.drawString( Integer.toString( c ++ ), // postincrement
14                          25, 40 );
15              g.drawString( Integer.toString( c ), 25, 55 );
16
17              c = 5;
18              g.drawString( Integer.toString( c ), 25, 85 );
19              g.drawString( Integer.toString( ++c ), //preincrement
20                          25, 100 );
21              g.drawString( Integer.toString( c ), 25, 115 );
22          }
23      }

```

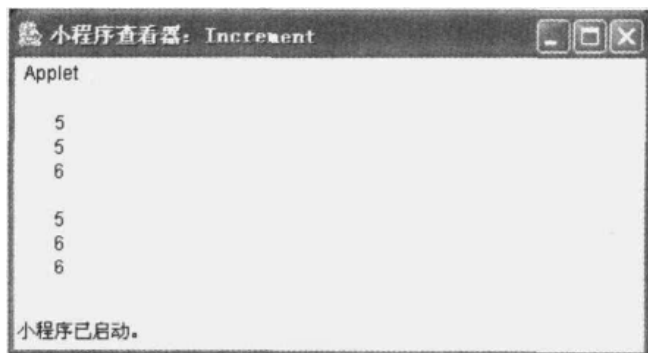


图 2.14 前置自增和后置自增操作的区别

该程序显示了 `c` 在使用 `++` 运算符之前和之后的值。自减运算符 (`--`) 的使用与此相类似。

#### 编程技巧 2.11

一元运算符应紧接着它的操作数，中间不要有空格。

图 2.11 中的三条赋值语句：

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

可以使用赋值运算符而更简洁地表示为：

```
passes += 1;
failures += 1;
student += 1;
```

若使用前置自增运算符，则表示为：

```
++ passes;
++ failures;
++ student;
```

若使用后置自增运算符，则表示为：

```
passes ++;
failures ++;
student ++;
```

在这里一定要注意，当在一条语句中只是将一个变量本身自增或自减时，那么前置自增和后置自增计算的结果是一样的，前置自减和后置自减计算的结果也是一样的。仅当一个变量出现在一个较大的表达式中时，对该变量进行前置自增和后置自增计算才会有不同的结果（前置自减和后置自减也同样）。

#### 常见编程错误 2.9

试图对一个表达式（而不是简单变量名）使用自增或自减运算符会导致语法错误。简单变量名是指可以出现在赋值表达式左边的变量名或表达式。比如，`++(x+1)` 是语法错误，因为 `(x+1)` 不是一个简单变量名。

图 2.15 中列出了前面介绍过的所有运算符的优先级顺序和结合性，各个运算符自顶向下按优

优先级递减的顺序排列。第二列中说明了相同优先级的运算符在计算时的结合性。注意, 条件运算符 (`?:`), 一元运算符中的自增 (`++`)、自减 (`--`)、正 (`+`)、负 (`-`) 和强制类型转换, 以及赋值运算符 (`=`、`+=`、`-=`、`*=`、`/=` 和 `%=`) 都是按从右向左的顺序结合, 所有其他运算符都是按从左向右的顺序结合。第三列中将为不同的“运算符组”命名。

运算符	结合性	类型
()	从左向右	括号
++ - --(type)	从右向左	一元
*/%	从左向右	倍数
+-	从左向右	相加
<<= >>=	从左向右	关系
== !=	从左向右	等于
?:	从右向左	条件
= += -= *= /= %=	从右向左	赋值

图 2.15 运算符的优先级顺序

## 2.13 基本数据类型

图 2.16 中列出了 Java 中的基本数据类型。基本数据类型用于构造更复杂的类型。Java 同 C 和 C++ 语言一样, 要求在使用程序中的所有变量之前必须对其进行类型声明。正因为如此, 我们将 Java 称为强类型语言。

类型	位长度	值	标准
boolean	1	true 或 false	
char	16	'\u0000' 至 '\uFFFF' (0~65 535)	(ISO Unicode 字符集)
byte	8	-128 至 +127 ( $-2^7 \sim 2^7 - 1$ )	
short	16	-32 768 至 +32 767 ( $-2^{15} \sim 2^{15} - 1$ )	
int	32	-2 147 483 648 至 +2 147 483 647 ( $-2^{31} \sim 2^{31} - 1$ )	
long	64	-9 223 372 036 854 775 808 至 +9 223 372 036 854 775 807	
float	32	-3.402 923 247E+38 至 +3.402 923 47E+38 ( $-2^{63} \sim 2^{62} - 1$ )	(IEEE 754 浮点)
double	64	-1.797 693 134 862 315 79E+308 至 +1.797 693 134 862 315 70E+308	(IEEE 754 浮点)

图 2.16 Java 的基本数据类型

与 C 和 C++ 不同, Java 中的基本数据类型在支持 Java 的所有计算机平台上都可以通用, 这就使程序员在编写程序时不会受到计算机平台的限制。

在 C 和 C++ 程序中, 程序员经常不得不多次编写同一个程序, 以支持多种不同的计算机平台, 这是因为基本数据类型在不同的计算机上可能是不一致的。例如, 在某一种计算机上的 int 值可能用 16 位 (或 2 个字节) 内存空间表示, 而在另一种计算机上 int 值可能用 32 位 (或 4 个字节) 内存空间表示。在 Java 中, int 值总是 32 位 (4 个字节)。

### 可移植性提示 2.1

与 C 和 C++ 语言不同, Java 中的基本数据类型在所有支持 Java 的计算机平台上都是通用的。Java 中的这些可移植特性, 使得程序员可以在不知道计算机平台的情况下编写程序。这一特性有时称为 WORA (Write Once Run Anywhere)。

在图 2.16 中, 对每个数据类型都列出了相应的位长度 (8 位是 1 个字节) 和值的范围。因为 Java

的设计者们想要使其具有可移植性，因此使用了国际标准的字符格式（Unicode）和浮点数格式（IEEE 754）。

## 2.14 常见的转义序列

在输出信息时，有时可以使用一个称为转义序列的特殊字符来添加一些格式。图 2.17 中列出了一些常见的转义序列。

在图 2.17 中，每个转义序列都以反斜杠（\）开头，反斜杠称为转义字符，表示将要输出一个“特殊”的字符。当一个反斜杠出现在一个字符序列中时，该反斜杠与其后面的字符组成了一个转义序列。

转义序列	描述
\n	换行符，将光标定位在下一行的开头
\t	垂直制表符，将光标移到下一个制表符的位置
\r	回车，将光标定位在当前行的开头；不会跳到下一行
\\	反斜杠，用于输出一个反斜杠字符
'	单引号，用于输出一个单引号字符
"	双引号，用于输出一个双引号字符

图 2.17 一些常见的转义序列

转义序列可以用在单个字符（括在单引号中）或字符串中的字符上。图 2.18 的应用程序说明了一些常见的转义序列的用法。第一条输出语句（第 7 行和第 8 行）使用两次“\”将一个字符括在一对单引号中。第二条输出语句（第 9 行和第 10 行）使用两次“\”将一个字符串括在一对双引号中。第三条输出语句（第 11 行）使用“\\”来输出一个反斜杠字符。第四条输出语句（第 12 行）使用两次“\t”输出一个中间插入了两个制表符的文本行，制表符可用于按列输出信息。第五条输出语句（第 13 行）两次使用“\n”输出具有双倍行距的文本，即两个文本行之间有一个空行。注意，System.out.println 方法与 System.out.print 方法实际上完成同样的功能，但前者会给输出的信息自动加上一个“\n”字符。最后一个输出语句使用了一个“\r”字符，说明光标重新定位在当前行的开始位置。注意，“#”字符覆盖了一些在“\r”字符之前输出的“\*”字符。

注意，只有使用 System.out 进行输出时才能使用转义序列 \n、\t 和 \r。

```
1 // Fig. 2.18: EscapeSequences.java
2 // Demonstrating common escape sequences
3
4 public class EscapeSequences {
5     public static void main( String args[] )
6     {
7         System.out.println( "Displaying single quotes: " +
8                             "'A'" );
9         System.out.println( "Displaying double quotes: " +
10                             "\"string\"" );
11         System.out.println( "Displaying a backslash: \\");
12         System.out.println( "Text separated\t\tby two tabs" );
13         System.out.println( "Here is double\n\nspaced text" );
14         System.out.println( "*****\r#####" );
15     }
16 }
```

**输出结果:**

```

Displaying single quotes: 'A'
Displaying double quotes: "string"
Displaying a backslash: \
Text separated          by two tabs
Here 's double

space text
#####

```

图 2.18 常见转义序列的用法

## 小结

- 由“可执行的操作”与“执行操作的顺序”所构成的解决问题的过程称为算法。
- 在计算机程序中指定语句的执行顺序称为程序控制。
- 伪代码可以帮助程序设计人员先“设想”出程序，然后再使用像 Java 这样的编程语言来写出程序。
- 声明是一些信息，用来告诉编译器变量的名字和属性，并让编译器为这些变量分配空间。
- 选择结构用于在许多操作中进行选择。
- if 选择结构只在条件为 true 时才执行操作。
- if/else 选择结构在条件为 true 和为 false 时可以指定执行不同的操作。
- 无论何时，如果想要像执行单个语句那样执行多条语句，就必须将它们括在一对花括号中以形成复合语句。复合语句可以放在任何单个语句放置的地方。
- 空语句表示不执行任何操作，句中只包含一个分号 (;)。
- 循环结构用于指定在某个条件保持为 true 时重复执行某个操作。
- while 循环结构的格式为：

```

while ( condition )
    statement

```

- 一个应用程序从 main 方法开始执行，应用程序使用 Java 解释器 (java) 来执行。
- main 方法的第一行语句必须是：

```
public static void main ( String args [ ] )
```

如果 main 方法不是以该语句开头，那么 Java 解释器将无法执行这个程序。

- 异常是指程序执行过程中出现的意外情况，它们不是经常发生的，但是有可能会发生。程序可以处理这些异常，从而使程序能够继续执行下去；也可以忽略这些异常，这时程序将会终止。
- Java 对于异常处理的要求特别严格。如果编译一个包含键盘输入的程序，那么 Java 会期望程序中能够处理可能发生的“输入异常”。如果想忽略这些异常，就必须向编译器指明，这是通过在方法的 throws 子句中声明异常来完成的。
- 标准输出流对象 System.out 用于向标准输出流（通常与屏幕相“联系”）输出信息。
- System.out.print 方法的功能是将其参数输出到屏幕上以显示出来。使用该方法时，要输出的字符串有时会放到一个缓冲区中（计算机内存中的一个临时存放区），而并不是显示到屏



幕上。System.out.flush 方法将强制显示缓冲区中的字符。

- System.out.println 方法的功能是将其参数输出到屏幕上以显示出来,然后再输出一个换行符。
- 标准输入流对象 System.in 用于获取来自标准输入流(通常是键盘)的输入。System.in.read 方法用于从键盘上读入一个字符,并返回计算机中表示该字符的整数值, System.in.skip 用于跳过输入流中的字符。
- 可以使用 Java 解释器来执行用 Java 编写的应用程序。如果要运行 Java 解释器来执行某个应用程序,只需在命令行上输入 java, 后面跟着 main 方法所在的类的名字,然后按下回车键即可。
- 带有小数部分的数称为浮点数,使用数据类型 float 或 double 表示。
- 一元的强制类型转换运算符用于将其操作数转换成一个浮点数。
- Java 提供了算术赋值运算符 +=、-=、\*=、/= 和 %=,用于简化某些常见类型的表达式。
- 自增运算符(++) 和自减运算符(--) 分别表示对一个变量加 1 和减 1。如果该运算符放在一个变量的前面,那么就首先将变量加 1 或减 1,然后再在表达式中使用该变量。如果该运算符放在一个变量的后面,那么就先在表达式中使用该变量,然后再将其加 1 或减 1。
- 基本数据类型(boolean、char、byte、short、int、long、float、double)用于构造 Java 的更复杂的数据类型。
- Java 要求在使用程序中的所有变量之前必须声明其类型。正因为如此,我们将 Java 称为强类型语言。
- Java 中的基本数据类型在支持 Java 的所有计算机平台上都可以通用。
- Java 使用了国际标准字符格式(Unicode)和浮点数格式(IEEE 754)。
- char、byte、short、int、long、float 和 double 类型的实例变量的默认初始值为 0, boolean 类型的变量的默认初始值为 false。

## 术语

-- operator	-- 运算符	counter-controlled repetition	计数器控制循环
++operator	++ 运算符	decision	判断
?:operator	?:运算符	decrement operator(--)	自减运算符(--)
action	动作	definite repetition	确定循环
action/decision model	操作 / 判断模式	double	
algorithm	算法	double-selection structure	双选择结构
application	应用程序	empty statement (;)	空语句(;)
arithmetic assignment operators:+=, -=, *=, /=		exception	异常
and %=	算术赋值运算符: +=, -=, *=, /= 和 %=	if selection structure	if 选择结构
block	块	if/else selection structure	if / else 选择结构
body of a loop	循环体	implicit conversion	隐式转换
cast operator	强制类型转换运算符	increment operator(++)	自增运算符(++)
compound statement	复合语句	indefinite repetition	不确定循环
conditional operator(?:)	条件运算符(?:)	infinite loop	无限循环
control structure	控制结构	initialization	初始化
		integer division	整数除法

IOException	single-entry/single-exit control structures 单入口 / 单出口控制结构
ISO Unicode character set ISO Unicode 字符集	single-selection structure 单选择结构
keyword 关键字	stacked control structures 堆栈的控制结构
logic error 逻辑错误	standard input stream 标准输入流
loop counter 循环计数器	standard output stream 标准输出流
loop-continuation condition 循环条件	structured programming 结构化编程
main method main 方法	syntax error 语法错误
nested control structures 嵌套的控制结构	System.in
newline character 换行符	System.in.read
nonfatal error 非致命错误	System.in.skip
postdecrement operator 后置自减运算符	System.out
postincrement operator 后置自增运算符	System.out.flush
predecrement operator 前置自减运算符	System.out.print
preincrement operator 前置自增运算符	System.out.println
promotion 提升	top-down, stepwise refinement 自顶向下、逐步求精
pseudocode 伪代码	unary operator 一元运算符
repetition 循环	while repetition structure while 循环结构
repetition structures 循环结构	whitespace characters 空白字符
selection 选择	
sentinel value 标志值	
sequential execution 顺序执行	

## 自测练习

### 2.1 填空:

- 所有的程序都可以按照三种控制结构: \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_来编写。
- \_\_\_\_\_选择结构在某个条件为 true 时执行一个操作, 而该条件为 false 时则执行另一个操作。
- 重复执行固定次数的循环称为\_\_\_\_\_循环。
- 当预先不知道一个语句序列将要重复执行多少次时, 就使用\_\_\_\_\_值来终止循环。

### 2.2 写出给变量 x 加 1 的 4 种 Java 语句。

### 2.3 写出完成下列功能的 Java 语句:

- x 和 y 相加, 将结果赋给 z, 然后再将 x 的值加 1。
- 判断变量 count 的值是否大于 10。如果大于 10, 就打印 "Count is greater than 10"。
- 将变量 x 减 1, 然后再从变量 total 中减去该值。
- 计算 q 除以 divisor 的余数, 并将结果赋给 q。使用两种方法写出该语句。

### 2.4 写出一条 Java 语句, 完成下列功能:

- 将变量 sum 和 x 声明为 int 类型。
- 将 1 赋给变量 x。
- 将 0 赋给变量 sum。
- 将变量 x 和 sum 相加, 并将结果赋给变量 sum。

- e) 打印 "The sum is: " 和变量 sum 的值。
- 2.5 将上一个练习中写出的各条语句组合到一个应用程序中, 计算并打印从 1 相加到 10 的整数和。使用 while 结构循环执行相加计算和加 1 操作。当 x 的值变为 11 时循环将会终止。
- 2.6 确定下列计算完成之后每个变量的值。假设每条语句开始执行时各变量的值都为整数 5。
- a) `product *= x++;`  
b) `quotient /= ++x;`
- 2.7 找出并改正下列语句中的错误:
- a) `while( c <= 5 ){`  
    `product *= c;`  
    `++c;`
- b) `if( gender == 1 )`  
    `System.out.println ( "Woman" );`  
    `else;`  
    `System.out.println ( "Man" );`
- 2.8 下面的 while 循环结构有什么错误:
- ```
while( z >= 0 )
    sum += z ;
```

## 自测练习答案

- 2.1 a) 顺序结构, 选择结构, 循环结构。  
b) if/else。  
c) 计数器控制或确定。  
d) 标志。
- 2.2 `x = x+1 ;`  
`x+ = 1 ;`  
`++x ;`  
`x++ ;`
- 2.3 a) `z =x+++ y ;`  
b) `if ( count > 10 )`  
    `System.out.println ( "Count is greater than 10" );`  
c) `total -= - -x ;`  
d) `q %= divisor ;`  
    `q = q % divisor ;`
- 2.4 a) `int sum, x;`  
b) `x = 1 ;`  
c) `sum = 0 ;`  
d) `sum += x ;`或 `sum = sum + x;`  
e) `System.out.println( "The sum is :" + sum );`
- 2.5 程序如下:
- ```
// Calculate the sum of the integers from 1 to 10
public class Calculate{
```

```

public static void main( String args [] )
{
    int sum, x ;
    x = 1 ;
    sum = 0 ;
    while( x <= 10 ){
        sum += x ;
        ++ x ;
    }
    System.out.println ( " The sum is : " + sum ) ;
}

```

- 2.6 a) product = 20, x = 6 ;  
 b) quotient = 3, x = 6 ;
- 2.7 a) 错误: 丢失 while 结构体的右花括号。  
 改正: 在语句 “++c;” 后面加上右花括号。  
 b) 错误: else 后面的分号会导致一个逻辑错误。第二条输出语句无论何时都会执行。  
 改正: 去掉 else 后面的分号。
- 2.8 在 while 结构中一直没有修改变量 z 的值。因此, 如果开始时循环条件 (z >= 0) 为 true, 就会产生一个无限循环。为了避免无限循环, z 必须不断递减, 这样它才会最终变成小于 0 的值。

## 练习

- 2.9 找出并改正下列语句中的错误 (注意, 每段代码中可能包含多个错误):

```

a) if ( age >= 65 ) ;
    System.out.println ( "Age greater than or equal to 65 " ) ;
    else
        System.out.println ( " Age is less than 65 ) " ;
b) int x = 1 , total ;
    while( x <= 10 ){
        total += x ;
        ++x ;
    }
c) While( x <= 100 )
    total += x ;
    ++x ;
d) while( y > 0 ){
    System.out.println ( y ) ;
    ++y ;
}

```

- 2.10 下面程序的输出结果是什么?

```

public class Mystery{
    public static void main( String args [])

```

```

    {
        int y, x = 1, total = 0 ;
        while( x <= 10 ) {
            y = x * x ;
            System.out.println ( y ) ;
            Total += y ;
            ++x ;
        }
        System.out.println ( "Total is"+ total ) ;
    }
}

```

按照下面的步骤完成练习 2.11 到练习 2.14:

- a) 分析并理解有问题的语句。
  - b) 使用伪代码和自顶向下、逐步求精的方法来构造算法。
  - c) 编写 Java 程序。
  - d) 测试、调试并运行该 Java 程序。
- 2.11 由于汽油价格昂贵, 因此司机们很关心他们的汽车每消耗 1 加仑汽油所能行驶的里程数。一名司机一直在记录每消耗一桶汽油所行驶的里程数及该桶汽油的加仑数。现在开发一个 Java applet, 接收每桶汽油的行驶里程数及该桶汽油的加仑数 (都是整数) 作为输入。运行该程序, 并输入这个司机所记录的值, 计算并显示每使用一桶汽油所得到的“消耗 1 加仑汽油行驶的里程数”, 并打印出平均的“消耗 1 加仑汽油行驶的里程数”(根据所有汽油的使用情况来计算平均值)。所有的平均值计算应采用浮点数结果。使用两个文本字段来输入数据 (提示: 参见图 1.13)。
- 2.12 开发一个 Java applet, 确定一个商店的客户是否已超过了其信用卡的透支数。对于每个客户应提供下列信息:
- a) 账号
  - b) 本月初的余额
  - c) 该客户本月所购置的所有商品的总额
  - d) 本月该客户账号中的存款总额
  - e) 允许的信用卡限额
- 本程序应从文本字段中输入上述值 (整数值), 并计算新的余额 (一月初余额 + 付款 - 存款), 然后显示新的余额, 并判断该余额是否超过了该客户的赊购限额。对于超过了赊购限制的客户, 应显示信息 “Credit limit exceeded”。
- 2.13 一个大公司以佣金的方式给售货员付酬。售货员每周的酬金是 \$200 加上本周销售总额的 9%, 例如, 一个售货员一周中售出了价值 \$5 000 的商品, 那么他本周的酬金就是 \$200 加上 \$5 000 的 9%, 即 \$650。现在有一份每个售货员所售商品的列表, 这些商品的价格如下所示:

名称	价格
A	239.99
B	129.75
C	99.95
D	350.89

开发一个 Java 应用程序，输入一个售货员上周所售出的商品数目，然后计算并打印该售货员的收入。对一个售货员所售出的商品的数目没有限制。

- 2.14 开发一个 Java applet，计算每个雇员的总收入。公司对于每个雇员前 40 个小时的工作时间按正常标准付酬，而对于超过 40 个小时以上的工作时间则按 1.5 倍的标准付酬。现在有一份公司所有雇员的名单，名单中列出了上周每个雇员的工作时间，以及每小时的酬金。在程序中应先输入每个雇员的这些信息，然后确定并显示该雇员的总收入。使用文本字段输入数据。

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00):10.00
Salary is $390.00
```

```
Enter hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00):10.00
Salary is $400.00
```

```
Enter hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00):10.00
Salary is $415.00
```

```
Enter hours worked (-1 to end): -1
```

- 2.15 在计算机应用程序中，经常要遇到“寻找最大值”的过程（即找出一组值中的最大值）。例如，在一个确定销售比赛冠军的程序中，应先输入每个售货员所销售的商品数日，然后找出销售商品数目最多的那个售货员，他就是比赛的胜者。编写一个伪代码程序，然后再改写成 Java 程序，完成下列功能：输入 10 个数字，找出并打印最大的数字。提示，程序中应使用下面三个变量：

counter: 累计到 10 的计数器（即用于记录已输入了多少个数字，并判断何时处理完 10 个数字）。

number: 保存当前输入的数字。

largest: 保存到目前为止的最大值。

- 2.16 编写一个 Java 应用程序，利用循环过程打印下表：

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

- 2.17 使用类似练习 2.15 的方法找出 10 个数中的两个最大值。可以将所有的值一次输入。
- 2.18 修改图 2.11 的程序，验证它的输入。对于每一次输入进行判断，如果所输入的值不是 1 或 2，就继续循环，直到用户输入了一个正确的值。
- 2.19 下面的程序将输出什么结果？

```
public class Mystery2 {
    public static void main ( String args[] )
```

```

    {
        int count = 1 ;
        while ( count <= 10 ){
            System.out.println (count * 2 == 1 ? " **** ":"+++++++");
            ++ count ;
        }
    }
}

```

2.20 下面的程序将输出什么结果?

```

public class Mystery3 {
    public static void main ( String args[])
    {
        int row = 10,column ;
        while ( row >= 1 ) {
            column = 1 ;
            while ( column <= 10 ) {
                System.out.print( row * 2 == 1 ? " < " : " > " ) ;
                ++ column ;
            }
            -- row ;
            System.out.println ( ) ;
        }
    }
}

```

2.21 (悬挂 else 问题) 分别确定当 x 等于 9 且 y 等于 11、x 等于 11 且 y 等于 9 时下列程序的输出结果。注意, 编译器会忽略 Java 程序中的缩进。而且, Java 编译器总是将 else 与离它最近的 if 匹配, 除非使用花括号 ({} ) 指定不同的匹配。因为只凭简单一看, 程序员可能无法确定 else 与哪一个 if 匹配, 所以称之为“悬挂 else”问题。在下面的程序中, 我们取消了缩进, 使这个问题更具有挑战性。(提示: 应用已学过的缩进形式。)

```

a) if ( x < 10 )
    if ( y > 10 )
        System.out.println ( "*****" ) ;
    else
        System.out.println ( "#####" ) ;
        System.out.println ( "$$$$$" ) ;

b) if ( x < 10 ){
    if ( y > 10 )
        System.out.println ( "*****" ) ;
    }
    else{
        System.out.println ( "#####" ) ;
        System.out.println ( "$$$$$" ) ;
    }
}

```

2.22 (另一个悬挂 else 问题) 修改下面的程序, 使它们产生不同的输出。使用适当的缩进方

式。除了插入花括号，不能做其他任何修改。编译器忽略 Java 程序中的缩进。在下面的程序中，我们取消了缩进，使这个问题更具有挑战性。注意：很可能不需要做任何修改。

```
if ( y == 8 )
if ( x == 5 )
System.out.println ( "EEEE" ) ;
else
System.out.println ( "####" ) ;
System.out.println ( "SSSS" ) ;
System.out.println ( "GGGG" ) ;
```

a) 假设  $x = 5, y = 8$ ，则产生下面的输出结果：

```
EEEE
SSSS
GGGG
```

b) 假设  $x = 5, y = 8$ ，则产生下面的输出结果：

```
EEEE
```

c) 假设  $x = 5, y = 8$ ，则产生下面的输出结果：

```
EEEE
GGGG
```

d) 假设  $x = 5, y = 7$ ，则产生下面的输出结果：（注意：else 后的三条输出语句都属于一个复合语句。）

```
####
SSSS
GGGG
```

2.23 编写一个 applet，读入一个正方形的边长，然后按这个长度打印一个以星号 (\*) 为边的空心正方形，在 applet 的 paint 方法中使用 drawString 方法来完成。程序中应限制正方形边长在 1 到 20 之间。例如，如果程序读入的边长为 5，则应打印以下图形：

```
*****
*   *
*   *
*   *
*   *
*****
```

2.24 回文是一种正读和反读都一样的数字或文本段。例如，下面几个 5 位数字都是回文：12321，55555，45554 和 11611。编写一个 applet，读入一个 5 位数，然后判断该数是不是一个回文。

2.25 编写一个 applet，输入一个只包含 0 和 1 的整数（即“二进制”整数），然后打印与该数等值的十进制数。（提示：使用“取模”和“除法”运算符，将“二进制”数的每一位从右向左一次一个地消掉，提取出 0 和 1。在十进制系统中，最右边数字的位值是 1，右边第二个数字的位值是 10，然后是 100，然后是 1000，依次类推；二进制系统与此类似，最右边的数字的位值是 1，右边第二个数字的位置值是 2，然后是 4，然后是 8，依次类



推。这样,十进数234就可分解为 $4*1+3*10+2*100$ 。二进制数1101转换成十进制数就是 $1*1+0*2+1*4+1*8=1+0+4+8=13$ 。)

2.26 编写一个应用程序,显示下面的图形样式:

```
*****
*****
*****
*****
*****
*****
*****
*****
```

程序中只能使用下列三种输出语句之一:

```
System.out.println ( " " );
System.out.println ( " " );
System.out.println ( ) ;;
```

注意,上面的第三条输出语句表示程序只输出一个换行符,从而使输出移到下一行上。

2.27 编写一个 applet, 在状态栏中不断地显示整数2的幂, 即2、4、8、16、32、64等。该循环不要终止(即应创建一个无限循环)。当运行此程序时, 会发生什么情况?

2.28 下面的语句有什么错误? 修改这条语句, 使它完成程序员本来要完成的功能:

```
System.out.println ( ++(x + y) ) ;
```

2.29 编写一个 applet, 读入三个由用户在文本字段中输入的非零值, 确定并打印它们是否可以代表一个三角形的三条边。

2.30 编写一个 applet, 读入三个非零整数值, 确定它们是否可以代表一个正三角形的三条边, 如果是则打印这三个数。

2.31 一个公司想要通过电话传送数据, 但他们担心电话会被窃听。所有的数据都按四位整数发送, 要求读者编写一个程序来加密这些数据, 以便数据能更安全地传送。在 applet 中, 先读入一个由用户在文本字段中输入的四位整数, 然后按下面的方法加密: 将每个数字换算成该位与7的和并用10求模; 然后将第一个数字与第三个数字交换, 再将第二个数字与第四个数字交换。最后打印加密后的整数。另外再编写一个 applet, 输入一个加密后的四位整数, 然后将该数还原。

2.32 一个非负整数  $n$  的阶乘写为  $n!$  (读做“ $n$ 的阶乘”), 定义为:

$$n! = n(n-1)(n-2) \cdots 1 \quad (n \text{ 的值大于等于 } 1),$$

$$n! = 1 \quad (n=0)$$

例如,  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , 结果为120。

a) 编写一个 applet, 从文本字段中读入一个非负整数, 计算并打印它的阶乘。

b) 编写一个 applet, 使用下面的公式估算数学常量  $e$  的值:

$$e = 1 + 1/1! + 1/2! + 1/3! + \cdots$$

c) 编写一个 applet, 使用下面的公式计算  $e^x$  的值:

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + \cdots$$

## 第3章 控制结构（二）

### 教学目标

- 学会使用 for 和 do/while 循环结构在程序中重复执行语句
- 理解使用 switch 选择结构进行多项选择
- 学会使用 break 和 continue 程序控制语句
- 学会使用逻辑运算符

### 3.1 简介

在编写一个解决特定问题的程序之前,有必要对问题进行大致了解,并且有一个精心策划的解决方案。在编写程序时,同样有必要了解可供使用的构造块的类型,并且要使用已证明过的程序构造原则。本章将在介绍Java控制结构的过程中,继续提供理论和结构化编程原则,这里将学到的技术适用于包括Java在内的大多数高级语言。当我们开始正式讨论面向对象的编程时,就会看到本章和上一章中所介绍的控制结构对构造和操作对象大有用处

### 3.2 计数器控制循环的实质

计数器控制循环要求:

1. 一个控制变量(或循环计数器)的名字。
2. 控制变量的初值。
3. 控制变量每次通过循环时的增量(或减量)。
4. 测试控制变量的终值的条件(即循环是否继续)。

考虑如图 3.1 所示的简单程序,该程序从 1 到 10 打印数字。下列声明语句:

```
int counter = 1;
```

命名了控制变量(counter),将其声明为一个整数,在内存中为其保留空间,并将其初值设为1。需要初始化的声明实际上是可执行语句。

---

```
1 // Fig. 3.1: WhileCounter.java
2 // Counter-controlled repetition
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class WhileCounter extends Applet {
7     public void paint( Graphics g )
8     {
```

```
9      int counter = 1;           // initialization
10     int yPos = 25;
11
12     while ( counter <= 10 ) {    // repetition condition
13         g.drawString( Integer.toString( counter ),
14                       25, yPos );
15         ++counter;              // increment
16         yPos += 15;
17     }
18 }
19 }
```

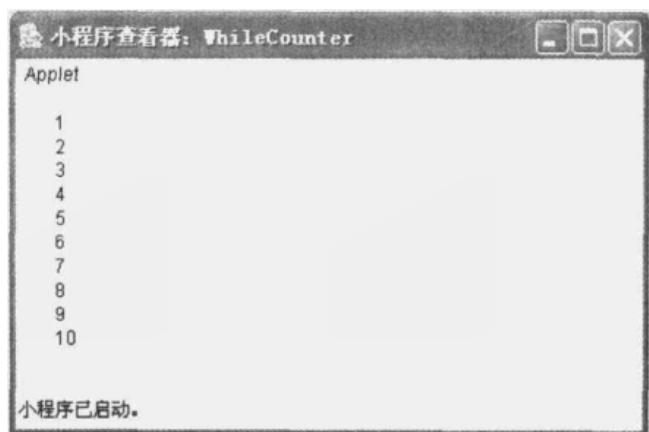


图 3.1 计数器控制循环

counter 的声明和初始化也可以使用下面的语句完成:

```
int counter;
counter = 1;
```

声明不可以执行,但赋值却可以。我们使用了两种方法对变量进行初始化。

下列语句:

```
++ counter;
```

使得循环每执行一次,循环变量就增加1。while结构中的循环条件测试控制变量的值是否小于或等于10(满足条件为true的最大值)。注意,在控制变量为10时也会执行此while结构;在控制变量超过10时(即counter变成11时)终止循环。

在图3.1中,通过将counter初始化为0,并使用下面的语句替换while结构而使程序变得更加简单。

```
while( ++ counter <= 10 ) { // repetition condition
    g.drawString ( Integer.toString ( counter ) , 25, yPos );
    yPos + = 15;
}
```

这里使用了更少的语句,因为在测试条件之前已经在while条件中直接完成了循环控制变量的增值。不过,使用这样的简写方式进行编程,需要拥有一定的实践经验。

#### 编程技巧 3.1

使用整型值来控制计数循环。

**常见编程错误 3.1**

由于浮点值可能不精确，使用浮点值控制计数循环可能会导致不准确的计数器值和不准确的终止测试。

**编程技巧 3.2**

在每个控制结构体中缩进语句。

**编程技巧 3.3**

在每个主要的控制结构前后各放置一空行，从而在程序中突出显示

**编程技巧 3.4**

过多的嵌套层次会使程序难以理解。作为一般规则，应尽量避免超过三级缩进。

**编程技巧 3.5**

在控制结构的上下留出的空间，并在控制结构首部缩进控制结构体，这样可以给程序一个二维的外观，大大增强了可读性。

### 3.3 for 循环结构

for 循环结构主要用来处理计数器的控制循环。为了说明 for 的作用，让我们重写图 3.1 的程序，如图 3.2 所示。

```
1      // Fig. 3.2: ForCounter.java
2      // Counter-controlled repetition with the for structure
3      import java.awt.Graphics;
4      import java.applet.Applet;
5
6      public class ForCounter extends Applet {
7          public void paint( Graphics g )
8          {
9              int yPos = 25;
10
11              // Initialization, repetition condition, and incrementing
12              // are all included in the for structure header.
13              for ( int counter = 1; counter <= 10; counter++ ) {
14                  g.drawString( Integer.toString( counter ), 25, yPos );
15                  yPos += 15;
16              }
17          }
18      }
```

图 3.2 使用 for 结构的计数器控制循环

当 for 结构开始执行时，控制变量 counter 初始化为 1。注意，counter 在 for 内部声明，因此它只在 for 结构体中可见。接着，检查条件 “counter <= 10”。因为 counter 的初值为 1，条件已满足，于是结构体语句打印出 counter 的值，即 1。变量 counter 在表达式 “counter++” 中增值，然后程序再次循环并进行测试。因为当前控制变量的值是 2，没有超过其终值，于是程序再次执行结构体语句。这一过程一直持续到控制变量 counter 增加到 11 时，这时循环条件的测试失败，因此终止循环，程序将接着执行 for 结构之后的语句（在这种情况下，相当于程序末尾的 return 语句）。

注意，图 3.2 中使用循环条件 “counter <= 10”。如果程序员误写为 “counter < 10”，则循环将

只执行9次。这是一个常见的逻辑错误，我们称之为差1错误（off-by-one error）。

#### 常见编程错误 3.2

在 while 或 for 结构的条件中使用不正确的关系运算符或不正确的循环计数器终值，会导致差1错误。

#### 编程技巧 3.6

在 while 或 for 结构的条件中使用终值和“<=”关系运算符将有助于避免差1错误。例如，对于一个用于打印1到10值的循环，循环条件应为“counter <= 10”或者“counter < 11”（也是正确的），而非“counter < 10”（产生差1错误）。

图3.3更加清楚地展示了图3.2中的for结构。注意，for结构说明了带有控制变量的计数器控制循环中的每一项。如果在for的循环体中有不止一条语句（如图3.2所示），则要用花括号来定义循环体。

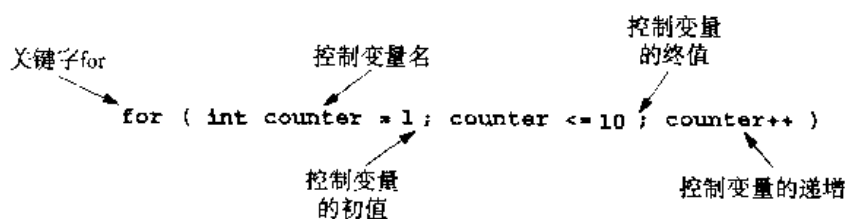


图 3.3 一个典型 for 结构的首部组成

for 结构的一般形式为：

```
for ( expression 1; expression 2; expression 3 )
    statement
```

其中，expression 1 初始化循环控制变量，expression 2 是循环条件，expression 3 增加控制变量的值。在多数情况下，for 结构可以使用以下等效的 while 结构来表示：

```
expression 1;
while ( expression 2 ) {
    statement
    expression 3;
}
```

在 3.7 节中，我们给出了一个不能用 while 语句替代的 for 结构。

如果使用包含在 for 首部结构内的 expression 1（初始化部分）来声明控制变量（即在控制变量名前指定其类型），则此控制变量只能在 for 结构内部使用。这种控制变量名的使用约束称为变量作用域（scope）。变量作用域是指程序中能够使用该变量的区域。例如，之前我们已经提到过，在程序中，只能使用在方法中声明的变量。关于变量作用域的详细内容，请参见第 4 章。

#### 常见编程错误 3.3

在 for 首部结构的开始定义控制变量后，如果在其结构之后使用该控制变量将导致语法错误。

有时候，expression 1 和 expression 2 在 for 结构中为逗号分隔的表达式列表，这使程序员能使用多重初始化表达式和多重增量表达式。例如，在一个 for 结构中可能必须初始化好几个控制变量。

### 编程技巧 3.7

在一个for结构的初始化和增量部分中只放置涉及控制变量的表达式。其他变量的操作或者出现在循环前（如果它们像初始化语句一样只执行一次）、或者出现在循环体中（如果它们像增量或减量语句一样每次循环都执行一次）。

for结构中的三个表达式是可选的。如果省略expression2，则Java假定循环条件为true，从而创建一个无限循环。如果在程序的其他地方初始化了控制变量，那么也可省略expression1。如果增量由for结构体中的语句计算或无需增量，则也可以省略expression3。for结构中的增量表达式就像for结构体末尾的一条单独语句。因此，表达式：

```
counter = counter + 1
counter += 1
++ counter
counter ++
```

在for结构的增量部分都是等效的。许多程序员更喜欢counter++形式，因为增值操作发生在循环体执行之后，所以后置递增形式似乎更自然。因为此处增值变量没有出现在一个表达式中，所以两种形式的增量都有相同的效果。for结构中的两个分号是必需的。

### 常见编程错误 3.4

在for首部结构中使用逗号而非分号。

### 常见编程错误 3.5

在for首部结构的右括号后放一个分号，使for结构体成为一条空语句，这通常是个逻辑错误。

在for结构的初始化、循环条件以及增量部分中可以包含数学表达式。例如，假定x=2和y=10。如果x和y在循环体中未修改，则下列语句：

```
for ( int j = x ; j <= 4 * x * y ; j += y/x )
```

等效于下列语句：

```
for ( int j = 2 ; j <= 80 ; j += 5 )
```

for结构的“增量”可以是负的（实际上是减量，循环向下计数）。

如果初始循环条件为false，那么就不会执行for结构体。取而代之的是执行for结构之后的语句。

控制变量常常用于打印或用于for结构体中的计算；但是也有例外，控制变量仅用于控制循环而在for结构体中从不使用的情况也是常见的。

### 测试和调试提示 3.1

尽管控制变量的值在一个for循环体中可以修改，但是因为这种情况会导致错误，所以要避免这样做。

for结构的流程图同while结构的流程图十分相似，例如：

```
for ( int counter = 1 ; counter <= 10 ; counter ++ ) {
    g.drawString ( Integer.toString ( counter ) , 25 , yPos ) ;
    yPos += 15;
}
```

如图3.4所示，这个流程图清楚地表明初始化只发生一次，而增值发生在每次结构体语句执行之后。注意（除了小圆圈和箭头符号之外），流程图只含有矩形符号和菱形符号。设想一下，程序

员可以到达空 for 结构的底层, 这和程序员使用其他控制结构的堆栈和嵌套来组成一个算法控制流的结构化实现完全一样。然后, 使用与该算法对应的动作和判断来填充矩形和菱形符号。

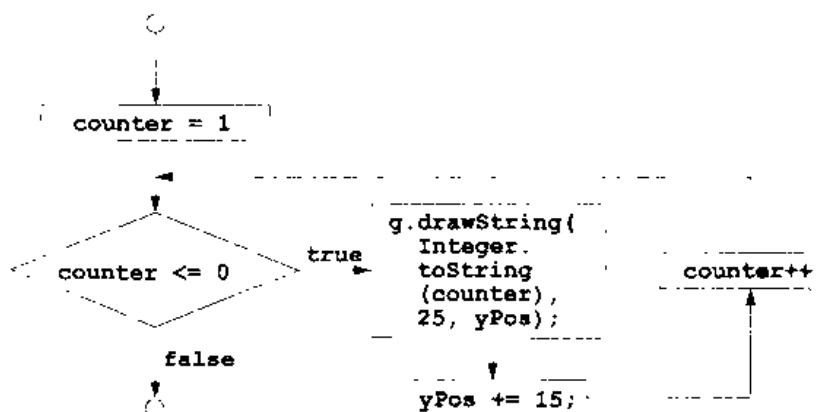


图 3.4 一个典型的 for 循环结构的流程图

### 3.4 使用 for 结构的例子

下面的例子表明在一个 for 结构中变化控制变量的方法。在每种情况下, 我们都编写了合适的 for 首部结构。请注意减少控制变量循环次数的关系运算符所发生的变化。

a) 以 1 为增量, 从 1 到 100 变化控制变量。

```
for ( int i = 1 ; i <= 100 ; i ++ )
```

b) 以 -1 为增量 (减量 1), 从 100 到 1 变化控制变量。

```
for ( int i = 100 ; i >= 1 ; i -- )
```

c) 以 7 为步长, 从 7 到 77 变化控制变量。

```
for ( int i = 7 ; i <= 77 ; i += 7 )
```

d) 以 -2 为步长, 从 20 到 2 变化控制变量。

```
for ( int i = 20 ; i >= 2 ; i -= 2 )
```

e) 以 2、5、8、11、14、17、20 的顺序变化控制变量。

```
for ( int j = 2 ; j <= 20 ; j += 3 )
```

f) 以 99、88、77、66、55、44、33、22、11、0 的顺序变化控制变量。

```
for ( int j = 99 ; j >= 0 ; j -= 11 )
```

#### 常见编程错误 3.6

在向下计数的循环条件中使用不正确的关系运算符 (例如在一个循环计数下降至 1 的循环中使用 `i <= 1`) 将导致逻辑错误, 并会在程序运行时产生不正确的结果。

下面两个例子演示了 for 循环结构的简单应用。

图 3.5 的程序使用了 for 结构来计算从 2 到 100 的所有偶数之和。注意, 图 3.5 中的 for 结构实际

上可以使用下面的逗号运算符而合并为 for 首部结构的最后部分:

```
for ( int number = 2 ; number <= 100 ;  
      sum += number , number += 2 )  
    ; // empty statement
```

同样, 初始化 `sum = 0` 也可以合并到 for 结构的初始化部分中。

```
1  // Fig. 3.5: Sum.java  
2  // Counter-controlled repetition with the for structure  
3  import java.awt.Graphics;  
4  import java.applet.Applet;  
5  
6  public class Sum extends Applet {  
7      public void paint( Graphics g )  
8      {  
9          int sum = 0;  
10  
11          for ( int number = 2; number<=100 ; number += 2 )  
12              sum += number;  
13  
14          g.drawString( "Sum is " + sum, 25, 25 );  
15      }  
16  }
```

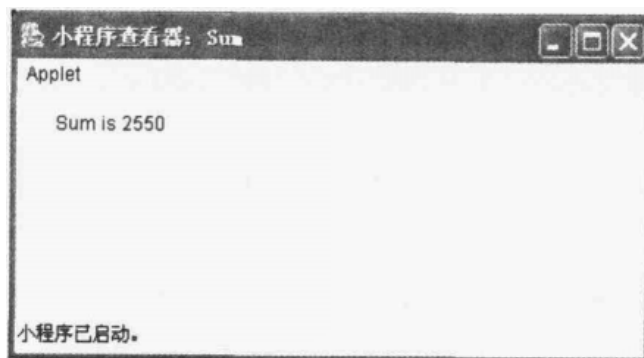


图 3.5 使用 for 求和

#### 编程技巧 3.8

尽管可以在 for 首部结构中将 for 语句之前的语句和 for 结构体中的语句合并, 但要避免这样做, 因为这会使程序更难读。

#### 编程技巧 3.9

如果有可能, 应将控制结构的首部限制在一行上。

下面的例子使用 for 结构来计算复利。考虑下面问题的陈述:

一个人向产生 5% 利息的储蓄账户投资了 \$100 000。假定所有的利息都留做储蓄, 计算并打印 10 年中每年年末的账户余额。使用下面的公式计算这些余额:

$$a = p(1+r)^n$$



其中:

$p$  是投入的原始数额 (即本金)

$r$  是年利率

$n$  是年数

$a$  是第几年年末的账户余额

解决这个问题的程序包含一个循环, 用于计算 10 年中每年用于储蓄的余额, 如图 3.6 所示。

```

1  // Fig. 3.6: Interest.java
2  // Calculating compound interest
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class Interest extends Applet {
7      public void paint( Graphics g )
8      {
9          double amount, principal = 1000.0, rate = .05;
10         int yPos = 40;
11
12         g.drawString( "Year", 25, 25 );
13         g.drawString( "Amount on deposit", 100, 25 );
14
15         for ( int year = 1; year <= 10; year++ ) {
16             amount = principal * Math.pow( 1.0 + rate, year );
17             g.drawString( Integer.toString( year ), 25, yPos );
18             g.drawString( Double.toString( amount ), 100, yPos );
19             yPos += 15;
20         }
21     }
22 }

```



图 3.6 使用 for 结构计算复利

for 结构执行 10 次循环体, 以 1 为增量从 1 到 10 改变控制变量。尽管 Java 没有指数运算符, 但我们可以使用 Math 类的方法 Math.pow 达到此目的。Math.pow(x,y) 方法计算  $x$  的  $y$  次幂, 它接收两个类型为 double 的参数并返回一个 double 值。double 类型是一个同 float 类型相似的浮点类型, 但是 double 类型的变量可以存储比 float 精度更高、量值更大的值。常量 (如图 3.6 中 1000.0 和 .05) 则被认为是 double 类型。

注意，我们声明变量 `amount`、`principal` 和 `rate` 为 `double` 类型，这样做仅仅是因为我们要处理美元的小数部分并且允许其值中含有小数点。不过，这样做也会导致一些问题。我们举个例子，解释使用 `float` 或 `double` 类型来表示美元数额时所引起的错误（假定美元数额以小数点后两位的精度显示）：两个在机器中存储的 `double` 美元数额可以是 14.234（为了显示方便一般四舍五入为 14.23）和 18.673（为了显示方便一般四舍五入为 18.67）。当这些数额相加时，它们产生的内部和为 32.907，为了显示方便，将会四舍五入为 32.91。因此打印输出应该如下所示：

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

但要按照以上打印的各数相加则会得到 32.90，显然会产生错误。

#### 编程技巧 3.10

不要使用 `float` 或者 `double` 类型的变量来执行精确的货币计算。浮点数的不精确会导致不正确的货币计算值。在练习中，我们使用整数来表示货币金额。注意：类库可以用来正确完成货币计算。

注意，作为 `Math.pow` 方法的一个参数，算式 “`1.0 + rate`” 包含在 `for` 结构体中。事实上，这个算式在每次循环时都将产生同样的结果，因此重复此计算是无意义的。

#### 性能提示 3.1

避免在循环内使用值不变的表达式。如果这样做了，许多较为先进的优化编译器会在编译所产生的机器语言代码中自动将这类表达式放在循环之外。

#### 性能提示 3.1

许多编译器都包含了一些代码的优化功能，不过最好还是从一开始就编写更精确的代码。

## 3.5 switch 多重选择结构

我们已经讨论了 `if` 单选择结构和 `if/else` 双选择结构。有时候，一个算法会包含一系列判断，其中的一个变量或表达式可以得出多个常量整型结果，并可根据这些结果而采取不同的动作。Java 提供了 `switch` 多重选择结构来处理此类判断过程。

`switch` 结构包含一系列 `case` 标号，还有可选的 `default` 情况。图 3.7 中的程序使用了 `switch` 结构，计算在考试中获得不同字母等级的学生人数。

```
1 // Fig. 3.7: SwitchTest.java
2 // Counting letter grades
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class SwitchTest extends Applet {
7     Label prompt;           // label for text field
8     TextField input;        // text field to enter grades
9
10    int aCount = 0, bCount = 0, cCount = 0,
11        dCount = 0, fCount = 0;
12}
```

```

13     public void init()
14     {
15         prompt = new Label( "Enter grade");
16         input = new TextField( 2 );
17         add( prompt );
18         add( input );
19     }
20
21     public void paint( Graphics g )
22     {
23         g.drawString( "Totals for each letter grade:", 25, 40 );
24         g.drawString( "A: " + aCount, 25, 55 );
25         g.drawString( "B: " + bCount, 25, 70 );
26         g.drawString( "C: " + cCount, 25, 85 );
27         g.drawString( "D: " + dCount, 25, 100 );
28         g.drawString( "F: " + fCount, 25, 115 );
29     }
30
31     public boolean action( Event e, Object o )
32     {
33         String val = o.toString();
34         char grade = val.charAt( 0 );
35
36         showStatus( "" );           // clear status bar area
37         input.setText( "" );        // clear input text field
38
39         switch ( grade ) {
40
41             case 'A': case 'a':      // Grade was uppercase A
42                 ++aCount;           // or lowercase a.
43                 break;
44
45             case 'B': case 'b':      // Grade was uppercase B
46                 ++bCount;           // or lowercase b.
47                 break;
48
49             case 'C': case 'c':      // Grade was uppercase C
50                 ++cCount;           // or lowercase c.
51                 break;
52
53             case 'D': case 'd':      // Grade was uppercase D
54                 ++dCount;           // or lowercase d.
55                 break;
56
57             case 'F': case 'f':      // Grade was uppercase F
58                 ++fCount;           // or lowercase f.
59                 break;
60
61             default:                 // catch all other characters
62                 showStatus( "Incorrect grade. Enter new grade." );
63                 break;
64         }
65
66         repaint(); // display summary of results
67         return true;
68     }
69 }

```

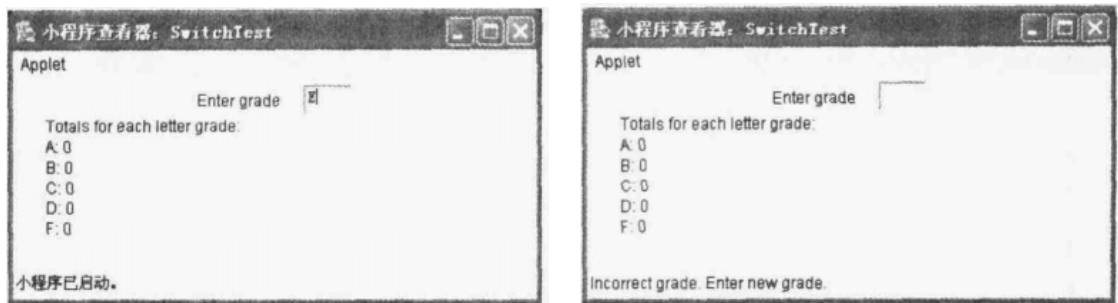


图 3.7 一个使用 switch 结构的例子

在此程序中，用户向文本字段中输入分类的字母等级。当按下回车键时，程序调用 action 方法以读取字母等级并进行处理。下列语句：

```
String val = o.toString ( );
```

接收用户在文本字段中输入的值并将其赋给 val。请记住，当用户按下回车键并且 o.toString() 允许我们在程序中将该值视为一个 String 时，action 方法自动接收文本字段中的数据（作为一个 String）。但是，我们实际上是想要处理用户输入的一个单独字符。下一行：

```
char grade = val.charAt ( 0 );
```

使用 charAt 方法选择该字符串中位置 0 的字符并将其赋给变量 grade，Java 认为一个字符串中的第一个字符在位置 0。charAt 方法可用于在一个字符串中选择任意位置的字符。例如，如果字符串 s 包含 “hello”，那么下列语句：

```
char letter = s.charAt (4);
```

将选择第五个字符（o）并将其赋给变量 letter。

在清除了文本字段和状态栏之后，输入 switch 结构。关键字 switch 后接括号内的变量名 grade 就是控制表达式。然后，使用该表达式的值与每个 case 标号进行比较。假定用户已输入了等级 C，C 将自动同 switch 中的每个 case 进行比较。如果有匹配（case 'C':），则执行这条 case 语句。对于字母 C，则会将 cCount 加 1，然后 switch 结构从 break 语句中退出。

break 语句导致程序控制转移到 switch 结构后的第一条语句上。使用 break 语句是因为一个 switch 语句中的每种 case 都会顺序执行。如果在一个 switch 结构中不使用 break，那么每次在结构中发生一次匹配时，所有余下的 case 语句都将执行。如果没有发生匹配，则执行 default 情况且打印错误消息。

每一个 case 都有多个动作。switch 结构不同于其他的一些结构，在 switch 结构的 case 中无需使用花括号将多个动作括起来。一般 switch 结构（在每个 case 中均使用一个 break）的流程图如图 3.8 所示。

流程图清楚地表明，每个 case 末尾的 break 语句将导致控制立即退出 switch 结构。请再次注意（除了小圆圈和箭头符号以外），该流程图只包含矩形和菱形符号。程序员能够到达空 switch 结构的底层，这和程序员使用其他控制结构的堆栈和嵌套来组成一个算法控制流的结构化实现完全一样。随后使用适合该算法的动作和判断来填充矩形和菱形符号。嵌套的控制是常见的，但在程序中很少能找到嵌套的 switch 结构。

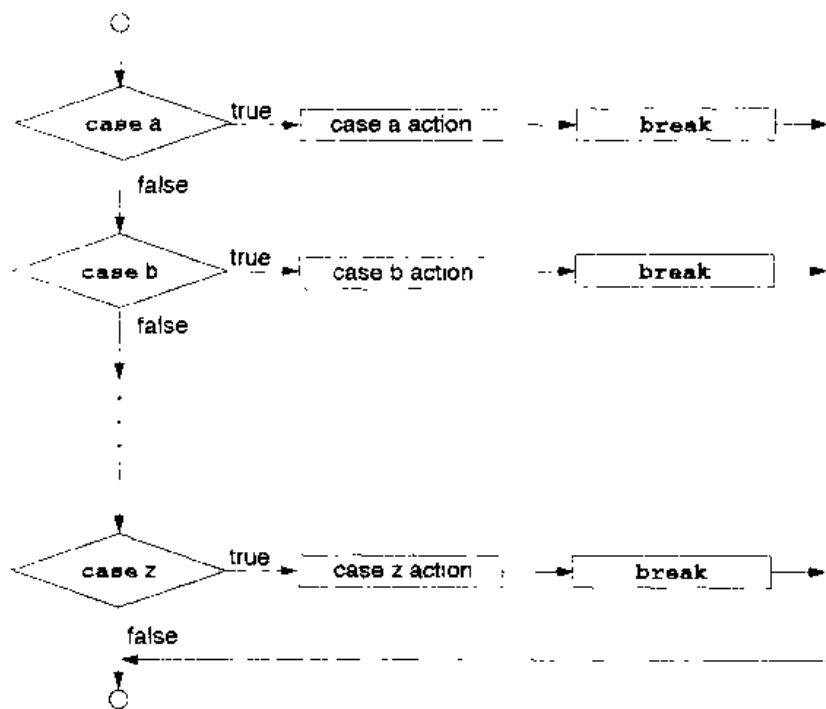


图 3.8 switch 多重选择结构

**常见编程错误 3.6**

在 switch 结构需要 break 语句时却忘记使用。

**编程技巧 3.11**

在 switch 语句中提供 default 情况。在没有 default 情况的 switch 语句中，没有显式测试的情况均被忽略。提供一个 default 情况会使程序员能注意到需要处理的异常情况。不过，有时也不需提供 default 处理。

**编程技巧 3.12**

尽管在 switch 结构中，case 子句和 default 子句可以按照任何顺序出现，但是作为一个良好的编程习惯，还是应将 default 子句放在最后。

**编程技巧 3.13**

在 switch 结构中，如果将 default 子句放在最后，则不再需要那条 case 子句的 break 语句。但为了清晰且与其他情况对称，一些程序员仍会加上 break 语句。

注意，列在一起的几个 case 标号简单地表示每一种情况都要执行同一组动作，例如“case 1:case 2:”。

当使用 switch 结构时，应记住在每个 case 之后的表达式只能是一个常量整型表达式，即任何字符常量的组合及能计算成一个常量整型值的整型常量。字符常量位于单引号中，如 'A' 就是一个字符常量。整型常量就是整型值。在 case 后的表达式还可以是一个常量变量（constant variable），即在整个程序中，其值保持不变的变量。这样的变量使用关键字 final 进行声明。当讨论面向对象的编程时，我们将提供更多实现 switch 逻辑的方式，我们将使用称为多态的技术，以创建比使用 switch 逻辑的程序更清晰、更易于维护和更易于扩展的程序。

## 3.6 do/while 循环结构

do/while 循环结构类似 while 结构。在 while 结构中，在循环体执行前的开始处测试循环条件。

do/while 结构则在循环体执行之后测试循环条件，因此循环体至少要执行一次。当一个 do/while 终止时，程序接着执行 while 子句后的语句。注意，如果在循环体中仅有一条语句，就没有必要在 do/while 结构中使用花括号。但是，花括号通常用来避免将 while 和 do/while 结构相混淆。例如：

```
while ( condition )
```

一般被认为是一个 while 结构的首部。如果在 do/while 结构中没有用花括号括起单条语句的结构体，则如下所示：

```
do
    statement
while ( condition );
```

这会引起混淆。最后一行“while ( condition );”会使读者误解成包含一个空语句的 while 结构。因此，为了避免混淆，含有一条语句的 do/while 常常写成如下形式：

```
do {
    statement
} while ( condition );
```

#### 编程技巧 3.14

一些程序员即使在无需使用花括号时也在 do/while 结构中使用花括号，这样可以避免将 while 结构同只有一条语句的 do/while 结构相混淆。

#### 常见编程错误 3.8

当 while、for 或 do/while 结构的循环终止条件永远不为 false 时，将会导致无限循环。为了防止这一问题，要保证一个 while 结构的首部后面没有分号。在一个计数器控制循环中，保证控制变量在循环中自增（或自减）。在一个标志控制循环中，保证输入标志值。

图 3.9 中的程序使用一个 do/while 结构来打印 1 到 10 的数字。注意，控制变量 counter 在循环条件测试中预先递增了。同时，我们使用花括号括起了 do/while 结构体中的单条语句。

```
1 // Fig. 3.9: DoWhileTest.java
2 // Using the do/while repetition structure
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class DoWhileTest extends Applet {
7     public void paint( Graphics g )
8     {
9         int counter = 1;
10        int xPos = 25;
11
12        do {
13            g.drawString( Integer.toString( counter ), xPos, 25 );
14            xPos += 15;
15        } while ( ++counter <= 10 );
16    }
17 }
```

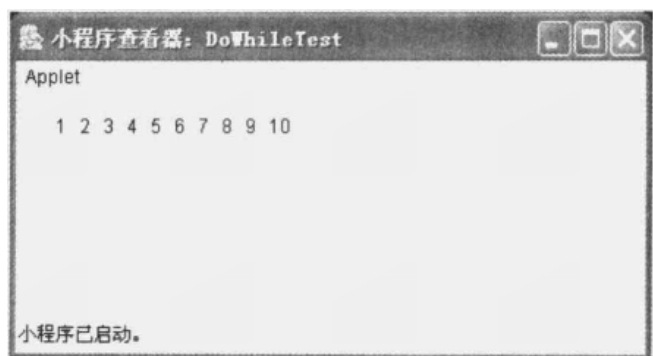


图 3.9 do/while 循环结构

do/while 流程图 (如图 3.10 所示) 清楚地表明, 程序至少完成一次动作之后才会测试循环条件。程序员可以到达 do/while 结构的底层, 这和使用其他控制结构的堆栈和嵌套来组成一个算法控制流的结构化实现完全一样。矩形和菱形符号由适合该算法的动作和判断条件进行填充。

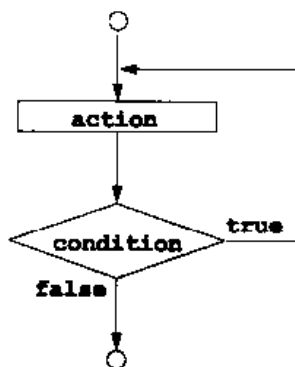


图 3.10 do/while 循环结构

### 3.7 break 和 continue 语句

break 和 continue 语句改变了控制流。当在 while、do/while 或 switch 结构中执行 break 语句时, 程序将立即从该结构中退出, 继续执行结构之后的语句。break 语句的常规用法是从一个循环中提前退出, 或者跳过一个 switch 结构的剩余部分 (如图 3.7 所示)。图 3.11 展示了一个用于 for 循环结构中的 break 语句。当 if 结构测得 count 已变成 5 时, 执行 break 语句, 这样就终止了 for 结构。程序将执行第二个 drawString, 即 for 之后的语句。程序中的循环完整地执行了 4 次。

```

1  // Fig. 3.11: BreakTest.java
2  // Using the break statement in a for structure
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class BreakTest extends Applet {
7      public void paint( Graphics g )
8      {
9          int count, xPos = 25;
10
11          for ( count = 1; count <= 10; count ++ ) {
12              if ( count == 5 )
  
```

```

13             break; // break loop only if count == 5
14
15             g.drawString( Integer.toString( count ), xPos, 25 );
16             xPos += 10;
17         }
18
19         g.drawString("Broke out of loop at count = " + count,
20                     25, 40 );
21     }
22 }

```

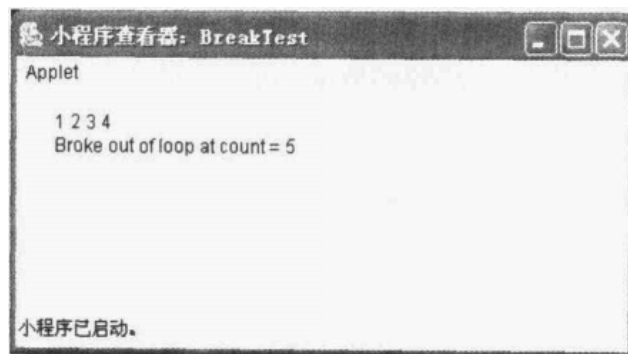


图 3.11 在一个 for 结构中使用 break 语句

当在 while、for 或 do/while 结构中执行 continue 语句时，会跳过该结构中的其余部分，继续执行下一次循环。在 while 和 do/while 结构中，在执行 continue 语句后执行测试循环条件的语句。在 for 结构中，执行增量表达式，然后执行循环条件的测试。在前面我们介绍过，while 结构在大多数情况下可替代 for 结构。但当 while 结构中的增量表达式跟在 continue 语句后时，则有一种例外情况。在这种情况下，增量操作在测试循环条件之前并未执行。在如图 3.12 所示的 for 结构中，当 count 为 5 时使用 continue 语句跳过了输出语句，并且开始了下一次循环。

```

1  // Fig. 3.12: ContinueTest.java
2  // Using the continue statement in a for structure
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class ContinueTest extends Applet {
7      public void paint( Graphics g )
8      {
9          int xPos = 25;
10
11          for ( int count = 1; count <= 10; count++ ) {
12              if ( count == 5 )
13                  continue; // skip remaining code in loop
14                          // only if count == 5
15
16              g.drawString( Integer.toString( count ), xPos, 25 );
17              xPos += 10;
18          }
19
20          g.drawString("Used continue to skip printing 5",
21                      25, 40 );
22      }
23  }

```



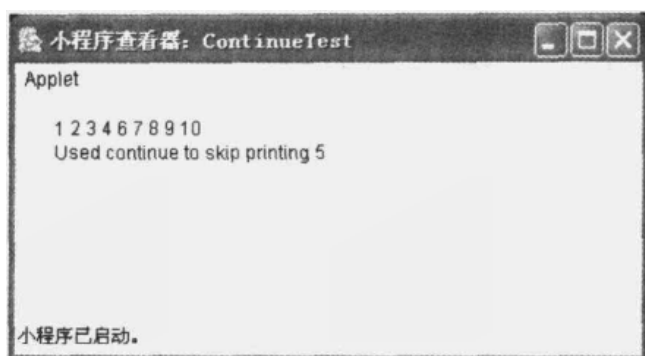


图 3.12 在一个 for 结构中使用 continue 语句

**编程技巧 3.16**

一些程序员觉得 break 和 continue 语句破坏了结构化编程。因为这些语句的作用效果也能使用我们将来学习的结构化编程技术来实现，所以这些程序员不使用 break 和 continue 语句。

**软件工程的观点 3.1**

在取得高质量软件工程和取得最佳软件性能之间存在着矛盾。常常是牺牲一方来达到另一方的目的。

**性能提示 3.3**

对于 break 和 continue 语句，如果使用得当，其运行速度将比相应的结构化技术更快。

## 3.8 带标记的 break 和 continue 语句

break 语句只能从封闭的 while、for、do/while 或 switch 结构中立即退出。为退出嵌套的结构集合，可以使用带标记的 break 语句。当在 while、for、do/while 或 switch 中执行该语句时，可从这些结构的任意多层封闭结构中立即退出；程序继续执行封闭的、带标记的复合语句（即一组包含在花括号中的语句，前面有一个标记）后的语句。图 3.13 演示了一个嵌套 for 结构中带标记的 break 语句。带标记的复合语句（第 11 行~第 28 行）以标记（后跟冒号标识符）开始，这里我们使用的是 stop。当第 18 行上的 if 结构检测到 row 等于 5 时，执行 break 语句。这条语句终止了第 16 行上的 for 结构和第 12 行上的封闭 for 结构。

```

1 // Fig. 3.13: BreakLabelTest.java
2 // Using the break statement with a label
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class BreakLabelTest extends Applet {
7     public void paint( Graphics g )
8     {
9         int xPos, yPos = 0;
10
11         stop: { // labeled compound statement
12             for ( int row = 1; row <= 10; row++ ) {
13                 xPos = 25;
14                 yPos += 15;
15
16                 for ( int column = 1; column <= 5 ; column++ ) {
17

```

```

18         if ( row == 5 )
19             break stop; // jump to stop label
20
21         g.drawString( "#", xPos, yPos );
22         xPos += 7;
23     }
24 }
25
26     yPos += 15;
27     g.drawString( "Loops terminated normally", 25, yPos );
28 }
29
30     yPos += 15;
31     g.drawString( "End of paint method", 25, yPos );
32 }
33 }

```

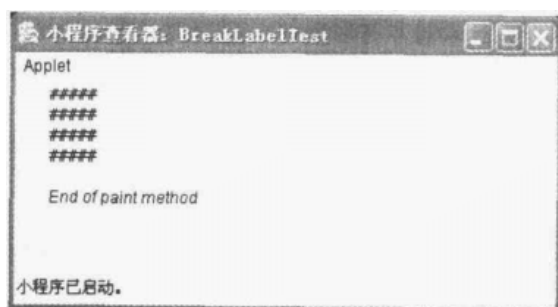


图 3.13 在嵌套 for 结构中使用了一个带标记的 break 语句

在第 30 行上继续执行，即带标记的复合语句后的第一条语句。内层的 for 结构只完整地执行了 4 次。注意，第 27 行上的 drawString 方法从未执行，因为它在带标记的复合语句中，且外层 for 结构从未完全执行。

continue 语句使封闭的 while、for 或 do/while 结构立即开始下一次迭代（循环）。当带标记的 continue 语句在一个循环结构（while、for 或 do/while）中执行时，会跳过结构体中的剩余部分和封闭循环结构，在执行 continue 语句后立即执行循环条件的测试。在带标记的 for 结构中，首先执行增量表达式，然后执行循环条件的测试。图 3.14 中的程序在嵌套 for 结构中使用带标记的 continue 语句，这样将执行外层 for 结构的下一次循环。带有标记的 for 结构（第 11 行 ~ 第 24 行）从 nextRow 标记处开始，当 for 结构内第 18 行的 if 结构检测到 column 大于 row 时，执行下列语句：

```
continue nextRow;
```

以及外层 for 循环的下一次迭代。即使内层 for 结构从 1 计数到 10，在一行上输出的 # 字符数也不会超过行数 row。

```

1 // Fig. 3.14: ContinueLabelTest.java
2 // Using the continue statement with a label
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class ContinueLabelTest extends Applet {
7     public void paint( Graphics g )
8     {

```

```

9      int xPos, yPos = 0;
10
11      nextRow:  // target label of continue statement
12      for ( int row = 1; row <= 5; row++ ) {
13          xPos = 25;
14          yPos += 15;
15
16          for ( int column = 1; column <= 10; column++ ) {
17
18              if ( column > row )
19                  continue nextRow; // jump to nextRow label
20
21              g.drawString( "#", xPos, yPos );
22              xPos += 7;
23          }
24      }
25  }
26  }

```

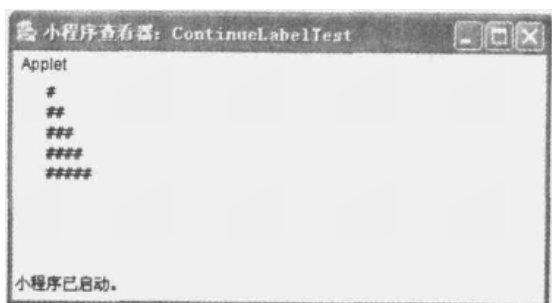


图 3.14 在嵌套的 for 结构中使用带标记的 continue 语句

### 3.9 逻辑运算符

到目前为止,我们只是研究了诸如“count <= 10”,“total > 1000”和“number != sentinelValue”这样的简单条件。这些条件由关系运算符>、<、>=、<=以及相等运算符==和!=来表达,每种判断都要基于一种条件。为了在判断过程中测试多个条件,必须在单独语句中或者在嵌套的if或if/else结构中执行这些测试。

Java提供了逻辑运算符,可以用来组合简单的条件以形成更复杂的条件。逻辑运算符包括:“&&”(逻辑与),“&”(布尔逻辑与),“||”(逻辑或),“|”(布尔逻辑或),“^”(布尔逻辑异或),以及“!”(逻辑非,即逻辑取反)。下面,我们分别给出每种运算符的例子。

假定我们希望在执行特定的路径之前,保证程序中的两个条件都为true。在这种情况下,我们可使用逻辑与运算符(&&),如下所示:

```

if ( gender == 1 && age >= 65 )
    ++seniorFemales;

```

这个if语句包含了两个简单条件。条件“gender == 1”可以用来判断一个人是否为女性,条件“age >= 65”用来判断一个人是否是老年公民。首先计算两个简单条件,因为“==”及“>=”的优先级均大于“&&”的优先级。接着考虑组合条件“gender == 1 && age >= 65”,当且仅当两个简单条件为

true时这个条件才为true。最后,如果这个组合条件的确为true,则seniorFemales的计数值增加1。如果两者之一或两个简单条件均为false,那么程序跳过下一条语句,执行if结构后的语句。对于上面的组合条件,可以通过添加冗余括号的办法来增强可读性,如下所示:

```
( gender == 1 ) && ( age >= 65 )
```

图3.15中的表格总结了逻辑与运算符,该表给出了表达式1和表达式2为false和true的4种组合。Java可计算出所有含有关系运算符、相等运算符和逻辑运算符的逻辑运算结果(false或true)。

表达式1	表达式2	表达式1&&表达式2
false	false	false
false	true	false
true	false	false
true	true	true

图 3.15 逻辑与运算符的真值表

现在让我们来看逻辑或运算符(II)。假定想在选择一条确定的执行路径之前,保证两个条件之一或两个条件都为true。那么在这种情况下,可使用逻辑或运算符,如下所示:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
    System.out.println ( " Student grade is A" );
```

这个语句也包含了两个简单条件。条件“semesterAverage >= 90”用来判断学生是否整个学期在这门功课上表现努力,因此可以获得一个“A”;条件finalExam用来判断学生是否在期末考试中取得优异成绩,因此可以获得一个“A”。接着,if语句考虑组合条件:

```
semesterAverage >= 90 || finalExam >= 90
```

如果两个简单条件之一或两个条件都为true,那么这名学生就为“A”级。注意,只有在两个简单条件均为false时才不打印消息“Student grade is A”。图3.16中给出了一个逻辑或运算符的真值表。

表达式1	表达式2	表达式1  表达式2
false	false	false
false	true	true
true	false	true
true	true	true

图 3.16 逻辑或运算符的真值表

逻辑与运算符比逻辑或运算符具有更高的优先级。这两个运算符都是从左向右结合的。一个包括逻辑与或是逻辑或运算符的表达式,只要其中一项的表达式为ture或false就不再继续计算了。因此,对于下面的表达式:

```
gender == 1 && age >= 65
```

如果gender不等于1(即整个表达式是false),那么计算就立即停止;如果gender等于1(即如果条件“age >= 65”为true,则整个表达式可以为true),则计算继续执行。这个性能特征称为短路计算(short-circuit evaluation)。

## 常见编程错误 3.9

如果表达式使用运算符“&&”，则一个条件（我们称其为依赖条件）可能需要另一个条件为 true 才有意义进行计算，这是有可能的。这种情况下，该依赖条件应当放在另一条件之后，否则可能出错。

## 性能提示 3.4

在使用运算符“&&”的表达式中，如果单独的条件彼此独立，则应让最有可能为 false 的条件处于最左边。在使用运算符“||”的表达式中，让最有可能为 true 的条件放在最左边，这样可以缩短程序的执行时间。

布尔逻辑与（&）和布尔逻辑或（|）运算符与常规逻辑与和逻辑或运算符基本一样，只有一个例外，即布尔逻辑运算符总是计算两个操作数（即没有短路计算）。因此，对于表达式：

```
gender == 1 & age >= 65
```

不管 gender 是否为 1 都要计算 age >= 65。如果布尔逻辑与（&）或布尔逻辑或（|）运算符的右操作数有一个必需的副作用——修改变量值，则这种计算方法就是有用的。例如，表达式：

```
birthday -= true | ++age >= 65
```

就将保证计算条件“++age >= 65”。因此，无论整个表达式是 true 还是 false，都将增加变量 age 的值。

含有布尔逻辑异或运算符（^）的表达式为 true 的条件是，当且仅当一个操作数为 true 且另一个操作数为 false。如果两个操作数均为 true 或 false，则整个条件的结果为 false。图 3.17 为逻辑异或运算符的真值表。这个运算符也要计算两个操作数（即不进行短路计算）。

表达式 1	表达式 2	表达式 1 ^ 表达式 2
false	false	false
false	true	true
true	false	true
true	true	false

图 3.17 布尔逻辑异或运算符的真值表

Java 提供了逻辑非运算符（!），使程序员能“反转”一个条件的含义。不同于逻辑运算符 &&、&、|| 和 |，它们组合了两个条件（二元运算符），逻辑非运算符只有一个条件作为操作数（一元运算符）。逻辑取反运算符放在一个条件之前，用来选择原始条件（没有逻辑非运算符）为 false 时的执行路径，如下列语句所示：

```
if ( ! ( grade == sentinelValue ) )
    System.out.println ( " The next grade is " + grade );
```

条件“grade == sentinelValue”外的括号是必需的，因为逻辑非运算符比相等运算符（==）的优先级更高。图 3.18 为逻辑非运算符的真值表。

表达式	!表达式
false	true
true	false

图 3.18 逻辑非运算符的真值表

在大多数情况下，程序员可以通过使用适当的关系运算符或相等运算符来表达条件，从而避免使用逻辑非运算符。例如，上述语句也可以写成：

```

if (grade != sentinelValue )
    System.out.println ( "The next grade is " + grade )

```

这种灵活性有助于程序员使用更简便的方式来表达条件。

图 3.19 的程序通过产生真值表而展示了所有逻辑运算符和布尔逻辑运算符。

在图 3.19 的输出中，字母 F 和 T 分别表示操作数的值为 false 和 true。条件的计算结果用 true 和 false 显示。

```

1 // Fig. 3.19: LogicalOperators.java
2 // Demonstrating the logical operators
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class LogicalOperators extends Applet {
7     public void paint( Graphics g )
8     {
9         g.drawString( "Logical AND (&&)", 10, 25 );
10        g.drawString( "F && F: " + ( false && false ), 10, 40 );
11        g.drawString( "F && T: " + ( false && true ), 10, 55 );
12        g.drawString( "T && F: " + ( true && false ), 10, 70 );
13        g.drawString( "T && T: " + ( true && true ), 10, 85 );
14
15        g.drawString( "Logical OR (||)", 215, 25 );
16        g.drawString( "F || F: " + ( false || false ), 215, 40 );
17        g.drawString( "F || T: " + ( false || true ), 215, 55 );
18        g.drawString( "T || F: " + ( true || false ), 215, 70 );
19        g.drawString( "T || T: " + ( true || true ), 215, 85 );
20
21        g.drawString( "Boolean logical AND (&)", 10, 115 );
22        g.drawString( "F & F: " + ( false & false ), 10, 130 );
23        g.drawString( "F & T: " + ( false & true ), 10, 145 );
24        g.drawString( "T & F: " + ( true & false ), 10, 160 );
25        g.drawString( "T & T: " + ( true & true ), 10, 175 );
26
27        g.drawString( "Boolean logical inclusive OR (|)",
28                    215, 115 );
29        g.drawString( "F | F: " + ( false | false ), 215, 130 );
30        g.drawString( "F | T: " + ( false | true ), 215, 145 );
31        g.drawString( "T | F: " + ( true | false ), 215, 160 );
32        g.drawString( "T | T: " + ( true | true ), 215, 175 );
33
34        g.drawString( "Boolean logical exclusive OR (^)",
35                    10, 205 );
36        g.drawString( "F ^ F: " + ( false ^ false ), 10, 220 );
37        g.drawString( "F ^ T: " + ( false ^ true ), 10, 235 );
38        g.drawString( "T ^ F: " + ( true ^ false ), 10, 250 );
39        g.drawString( "T ^ T: " + ( true ^ true ), 10, 265 );
40
41        g.drawString( "Logical NOT (!)",
42                    215, 205 );
43        g.drawString( "!F: " + ( !false ), 215, 220 );
44        g.drawString( "!T: " + ( !true ), 215, 235 );
45    }
46 }

```

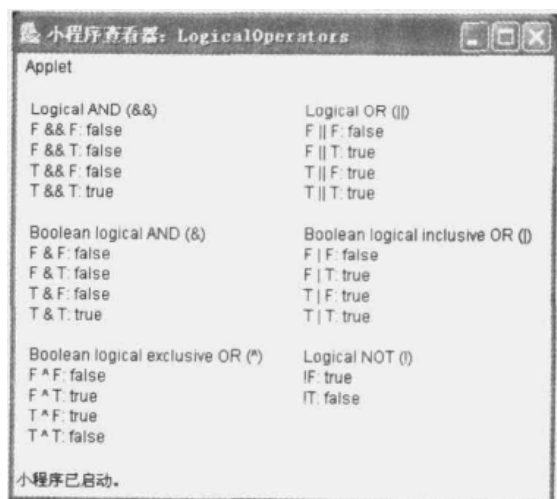


图 3.19 逻辑操作

图 3.20 中列出了目前引入的 Java 运算符的优先级和结合性。运算符以优先级递减的顺序自顶向下排列。

运算符	结合性	类型
( )	从左向右	括号
++ -- + - ! (type)	从右向左	一元运算符
* / %	从左向右	倍数
+ -	从左向右	加法
< <= > >=	从左向右	关系
== !=	从左向右	相等
&	从左向右	布尔逻辑与
^	从左向右	布尔逻辑异或
	从左向右	布尔逻辑或
&&	从左向右	逻辑与
	从左向右	逻辑或
?:	从右向左	条件
= += -= *= /= %=	从右向左	赋值

图 3.20 运算符优先级和结合性

### 3.10 结构化编程小结

就像建筑师用他们在本行业所积累的知识来设计建筑物一样,程序员也应使用所积累的知识来设计程序。编程领域比建筑学年轻得多,因此积累的知识也相当少。我们已经知道了使用结构化编程开发的程序比使用非结构化编程开发的程序更易于理解,因而更易于测试、调试、修改,并且在数学意义上更易于证明其正确性。

图 3.21 总结了 Java 的控制结构,小圆圈符号表示每个结构的单入口点和单出口点。随意连接单个的流程图符号会产生非结构化程序。因此,专业程序员选择组合流程图符号来形成控制结构的有限集合,并且以两种简单方式正确地组合控制结构来构造结构化程序。简而言之,即只使用单入口/单出口控制结构,其中只有一条通路进入且只有一条通路离开每个控制结构。

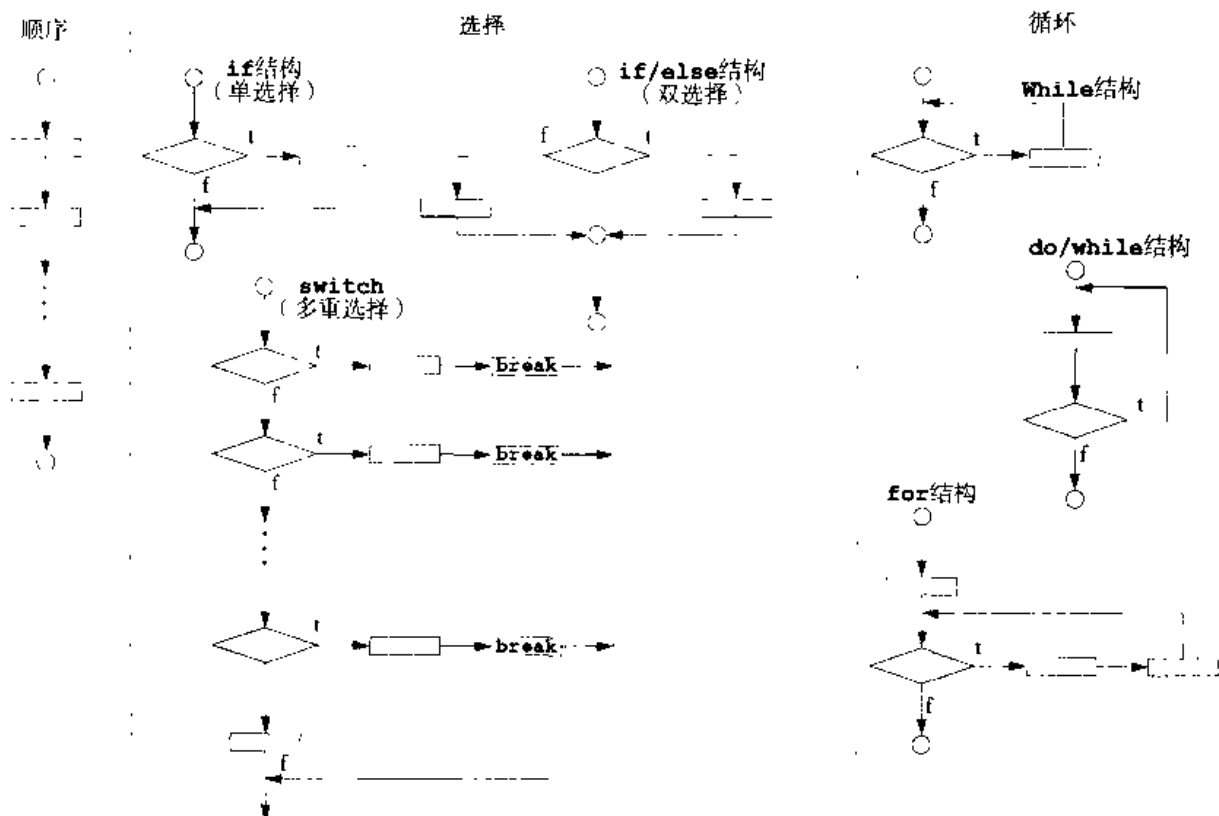


图 3.21 Java 的单入口 / 单出口的顺序、选择和循环控制结构

按顺序连接控制结构来组成结构化程序是很简单的,将一个控制结构的出口点连到下一个控制结构的入口点,即将程序中的控制结构简单地一个接一个排放在一起,我们称之为“控制结构堆栈”。组成结构化程序的规则也允许控制结构嵌套。

图 3.22 中给出了组成正确结构化程序的规则,图中假定矩形流程图符号可用于指明包括输入/输出在内的任何动作。

#### 组成结构化程序的规则

1. 以“最简流程图”开始(如图 3.23 所示)。
2. 任何矩形(动作)符号都可以使用顺序的两个矩形(动作)符号代替。
3. 任何矩形(动作)符号都可以使用任何控制结构(顺序、if、if/else、switch、while、do/while 或者 for)代替。
4. 规则 2 和规则 3 可以按照任何顺序应用任意多次。

图 3.22 组成结构化程序的规则

把图 3.22 中的规则应用到最简流程图(如图 3.23 所示)中,就能产生清晰的、具有构造块外观的结构化流程图。例如,在最简流程图中反复应用规则 2 会产生包括一串矩形的结构化流程图(如图 3.24 所示)。注意,规则 2 将产生控制结构堆栈,因此规则 2 称为堆栈规则(stacking rule)。

规则 3 称为嵌套规则。在最简流程图上应用规则 3 会产生清晰的嵌套控制结构。例如在图 3.25 中,首先使用一个双选择(if/else)结构替换最简单流程图中的矩形符号。接着再对双选择结构中的矩形符号应用一次规则 3,使用双选择结构替换每一个矩形符号。矩形符号周围的虚线框表示该矩形已经替换。

规则 4 产生了更大的、包含更多内容、嵌套更深的结构。在图 3.22 中应用这些规则产生的流程图构成了所有可能的结构化流程图集合,由此构成了所有可能的结构化程序的集合。

结构化方法的优势在于我们只使用了 7 个单入口/单出口块,并且使用 2 种方式进行组合。



图3.26给出了一些堆栈构造块（来自规则2的应用）和各种嵌套的构件块（来自规则3的应用）。图中也列出了不能出现在结构化流程图中的重叠构造块（因为没有 goto 语句）。

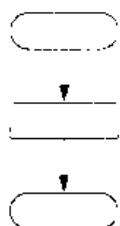


图 3.23 最简流程图

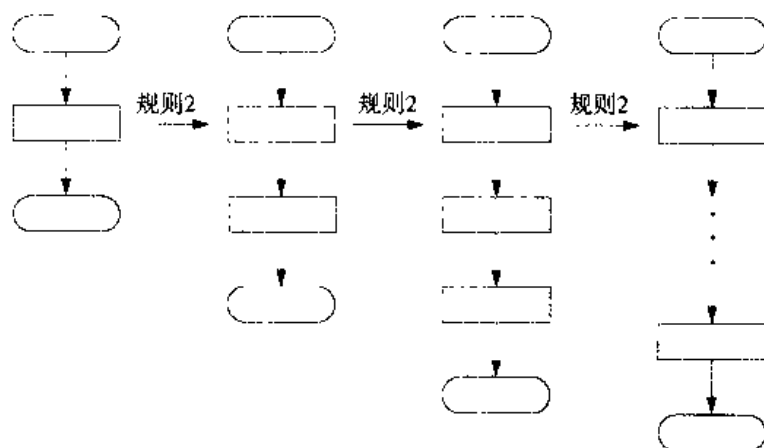


图 3.24 把图 3.22 的规则 2 重复应用到最简流程图

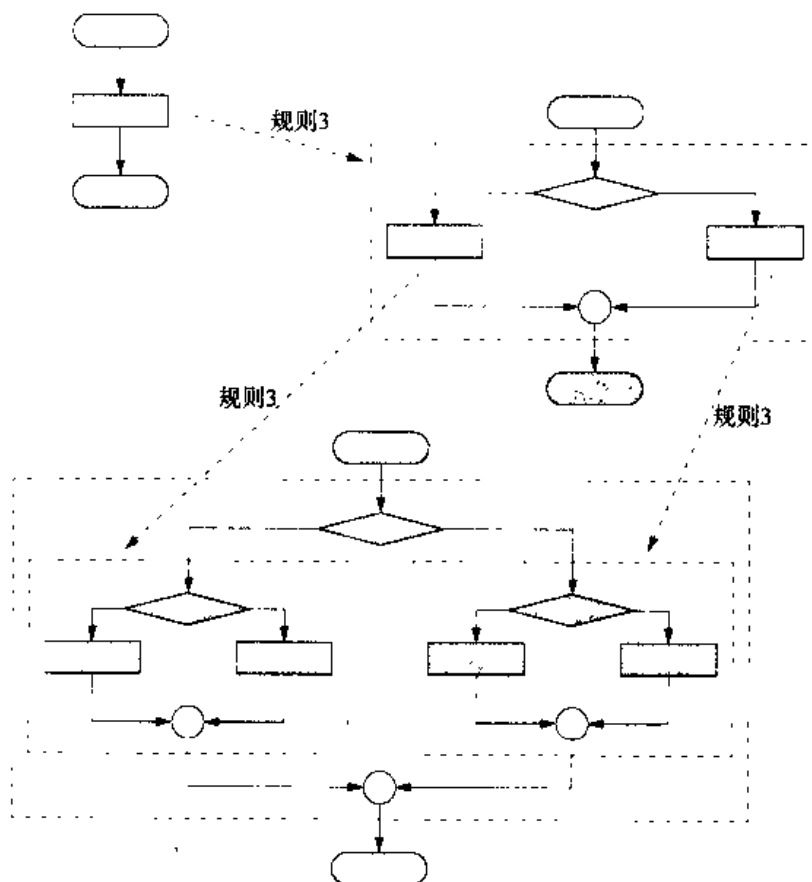


图 3.25 在最简流程图上应用图 3.22 的规则 3

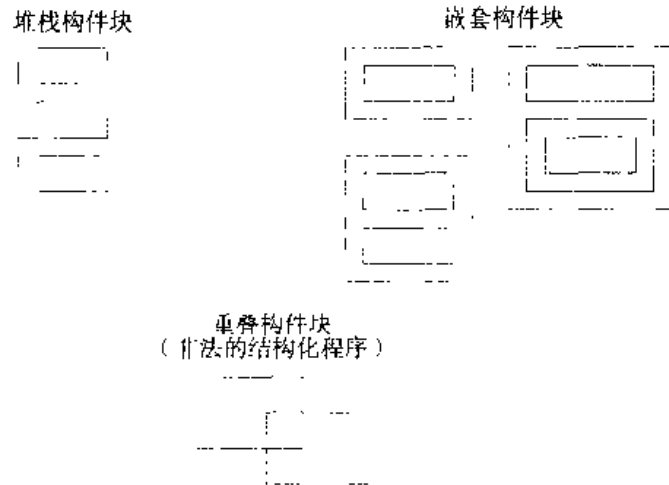


图 3.26 堆栈构件块、嵌套构件块及重叠构件块

如果遵守图 3.22 的规则, 就不会产生非结构化的流程 (如图 3.27 所示)。如果不能确定某一个流程图是否是结构化的, 那么不妨逆向应用图 3.22 中的规则来将流程图简化为最简流程图。如果流程图可以简化为最简流程图, 则原始流程图就是结构化的, 否则就不是。

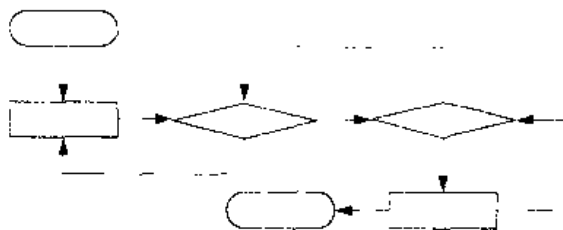


图 3.27 非结构化流程图

结构化编程提高了简洁性。事实上, 研究证明我们只需三种控制形式:

- 顺序
- 选择
- 循环

顺序控制是最基本的。选择控制可以使用下列三种方式之一实现:

- if 结构 (单选择)
- if/else 结构 (双选择)
- switch 结构 (多重选择)

事实上, 很容易证明简单的 if 结构足以提供任何形式的选择, 即任何可由 if/else 结构和 switch 结构实现的选择, 都可以由组合的 if 结构来实现。

循环控制可由下列三种方式之一实现:

- while 结构
- do/while 结构
- for 结构

while 结构足以提供任何形式的循环;任何可以由 do/while 结构和 for 结构实现的循环也可以由 while 结构实现(尽管有时不容易理解)。

综合这些结论,可以得知 Java 程序中所需的任何形式的控制都可以由下列方式实现:

- 顺序结构
- if 结构(选择)
- while 结构(循环)

而且这些控制结构可以只由两种方式(堆栈和嵌套)进行组合。事实上,结构化编程提高了程序的简洁性。

在本章中,我们讨论了如何从只包含有动作和判断的控制结构来产生程序。在第4章中,我们将详细讨论另一种称为方法(method)的程序结构单元(方法在其他编程语言中常常称为函数或过程)。我们将学习如何通过组合由控制结构形成的方法来编写大型程序,然后还将介绍 Java 的另一种程序结构单元,即类(class)。我们还将继续讨论面向对象的编程。

## 小结

- for 循环结构处理了计数器控制循环的所有细节。for 结构的一般格式为:

```
for (expression1; expression2; expression3 )
    statement
```

其中 expression1 用于控制变量的初始化,expression2 是循环条件,expression3 递增控制变量的值。

- do/while 循环结构在循环末尾测试循环条件,因此循环体至少执行一次。do/while 结构的格式为:

```
do {
    statement
} while (condition);
```

- break 语句在某一种循环结构(for、while 和 do/while)中执行时将导致从此结构中立即退出。
- continue 语句在某一种循环结构(for、while 和 do/while)中执行时将跳过该结构体中其余的语句,并继续执行下一次循环。
- switch 语句处理一系列判断,它测试某一个变量或表达式是否为其假定的值,然后采取不同的动作。在大多数程序中,有必要在每个 case 语句之后加上一个 break。几个 case 可以通过在语句之前一同列出 case 标号来执行相同的语句。switch 结构只能测试常量整型表达式。
- 逻辑运算符可用于通过条件组合来形成复合条件。逻辑运算符包括 &&、&、||、|、^ 和 !,分别表示逻辑与、布尔逻辑与、逻辑或、布尔逻辑或、布尔逻辑异或和逻辑非(取反)。

## 术语

&& operator    && 运算符

|| operator    || 运算符

! operator    !运算符

boolean logical AND(&)    布尔逻辑与(&)

boolean logical exclusive OR(^) 布尔逻辑异或(^)	logical negation(!) 逻辑非(!)
boolean logical inclusive OR(  ) 布尔逻辑或(  )	logical operators 逻辑运算符
break	logical OR(  ) 逻辑或(  )
case	long
continue	loop-continuation condition 循环条件
counter-controlled repetition 计数器控制循环	multiple selection 多重选择
default case in switch switch 的 default 情况	nested control structures 嵌套控制结构
definite repetition 有限循环	off-by-one error 差 1 错误
do/while repetition structure do/while 循环结构	repetition structures 循环结构
for repetition structure for 循环结构	short-circuit evaluation 短路计算
infinite loop 无限循环	single-entry/single-exit control structures 单入口/ 单出口控制结构
labeled break statement 带标记的 break 语句	stacked control structures 堆栈控制结构
labeled compound statement 带标记的复合语句	switch selection structure switch 选择结构
labeled continue statement 带标记的 continue 语句	while repetition structure while 循环结构
labeled repetition structure 带标记的循环结构	
logical AND(&&) 逻辑与(&&)	

## 自测练习

- 3.1 判断下列问题是否正确。如果不正确，请解释原因。
  - a) 在 switch 选择结构中需要 default 情况。
  - b) 在一个 switch 选择结构的默认情况中需要 break 语句。
  - c) 表达式“(x > y && a < b)”在 x > y 为 true 或 a < b 为 true 时为 true。
  - d) 一个包括“||”的表达式在其操作数之一或者两者皆为 true 时为 true。
- 3.2 使用一条或一组 Java 语句完成下列问题：
  - a) 使用一个 for 结构求出 1 到 99 之间所有数之和。假定整数变量 sum 和 count 已经声明。
  - b) 使用 pow 方法计算 2.5 的 3 次方。
  - c) 使用一个 while 循环和计数器变量 x 打印从 1 到 20 的整数。假定变量 x 已经声明，但未初始化。每行打印 5 个整数。提示：使用算式“x % 5”。当这个值为 0 时，打印一个换行符，否则打印一个制表符。假定这是一个应用，使用 System.out.println() 方法输出换行符，使用 System.out.print('t') 方法输出制表符。
- 3.3 在下面的程序段中找出错误并解释如何改正。
  - a) 

```
x = 1;
while ( x<= 10 ) ;
    x++;
}
```
  - b) 

```
for ( y = .1; y != 1.0; y+=.1 )
    System.out.println(y);
```
  - c) 

```
switch ( n ) {
    case 1:
        System.out.println( "The number is 1" );
```

```

        case 2:
            System.out.println( "The numner is 2" );
            break;

        default:
            System.out.println( "The number is not 1 or 2" );
            break;
    }
}
d) 下面的代码应打印 1 到 10 的值。
    n = 1;
    while ( n < 10 )
        System.out.println( n++ );

```

## 自测练习答案

- 3.1 a) 不正确。default 情况是可选的。如果无需默认动作，就不必使用一个 default 情况。  
 b) 不正确。break 语句用于退出 switch 结构。在 default 情况是最后一个情况时不需要 break 语句。  
 c) 不正确。使用 “&&” 运算符时，两个关系表达式必须同时为 true 才能使整个表达式为 true。  
 d) 正确。

- 3.2 a) 

```
sum = 0;
for ( count = 1; count <= 99; coount += 2 )
    sum += count;
```

  
 b) 

```
Math.pow( 2.5, 3 )
```

  
 c) 

```
x = 1;
```

```

while ( x<= 20 ) {
    System.out.print( x );

    if ( x % 5 == 0 )
        System.out.println();
    else
        System.out.print( '\t' );
    x++;
}

```

- d) 

```
for ( x =1; x <= 20; x++ ) {
    System.out.print( x );

    if ( x % 5 == 0 )
        System.out.println();
    else
        System.out.print( '\t' );
}
```

或者:

```
for ( x = 1; x <= 20; x++ )

    if ( x % 5 == 0 )
        System.out.println( x );
    else
        System.out.print( x + " " );
```

3.3 a) 错误: while 首部后的分号导致了一个无限循环。

改正: 用一个 “{” 替换分号, 或者删除 “:” 和 “}”。

b) 错误: 使用一个浮点数来控制一个 for 循环结构。

改正: 使用一个整数完成正确的计数, 从而获得想要的值。

```
for( y = 1 ; y != 10 ; y ++ )
    System.out.println( (float) y / 10 );
```

c) 错误: 第一条 case 语句中少了 break 语句。

改正: 在第一条 case 语句末尾加上一个 break 语句。注意, 如果程序员希望在每次执行 “case 1:” 的语句时都执行 “case 2:” 的语句, 那么就不算错误。

d) 错误: 用在 while 循环条件中的关系运算符不正确

改正: 用 “<=” 代替 “<”, 或把 10 改为 11。

## 练习

3.4 找出下列语句中的错误 (注意: 可能不止一个错误):

a) For ( x= 100, x >= 1, x++ )

```
System.out.println( x );
```

b) 下面的代码打印的整数 value 是奇数, 还是偶数:

```
switch ( value % 2 ) {

    case 0:
        System.out.println( "Even integer" );
    case 1:
        System.out.println( "Odd integer" );
}
```

c) 下面的代码应当输出从 19 到 1 的奇数:

```
for ( x = 39; x >= 1; x += 2 )
    System.out.println( x );
```

d) 下面的代码应当输出从 2 到 200 的偶数:

```
counter = 2;
do {
    System.out.println( counter );
    counter += 2;
} While ( counter < 100 );
```

3.5 下面程序的功能是什么?

```

public class Printing {
    public static void main (String args[] )
    {
        for ( int i = 1; i <= 10; i++ ) {

            for ( int j = 1; j <= 5; j++ )
                System.out.print( "@" );

            System.out.println();

        }
    }
}

```

- 3.6 编写一个应用程序，找出几个字母中最小的一个。假定第一个读入的值是一个说明其余字符个数的数字字符。
- 3.7 编写一个 applet 程序，计算并打印从 1 到 15 的奇数的乘积。
- 3.8 factorial 方法常常用于求解概率问题。一个正整数  $n$  的阶乘（写为  $n!$ ，读做“ $n$  阶乘”）等于从 1 到  $n$  的所有正整数的乘积。编写一个 applet 程序，计算整数 1 到 5 的阶乘，以表格形式显示结果。另外，在计算 20 的阶乘时会遇到什么困难？
- 3.9 修改图 3.6 中的复利程序，循环计算其利率分别为 5%、6%、7%、8%、9% 和 10% 的情况。使用一个 for 循环来改变利率。
- 3.10 编写应用程序，一个接一个地分别打印下列图案。使用 for 循环来产生这些图案，使用语句 “System.out.print(‘\*’)” 打印所有的星号（\*）（这会使星号一个接一个地打印）。提示：最后两种图案需要每行开头有适当数量的空格。额外要求：将 4 个独立问题的代码组成一个程序，并灵活应用 for 循环一个接一个地打印 4 种图案。

(a)	(b)	(c)	(d)
*	*****	*****	*
**	*****	*****	**
***	*****	*****	***
****	*****	*****	****
*****	*****	*****	*****
*****	*****	*****	*****
*****	****	****	*****
*****	***	***	*****
*****	**	**	*****
*****	*	*	*****

- 3.11 计算机的一个有趣应用是绘制曲线图和柱形图（有时称为柱状图）。编写一个 applet 程序，读入 5 个数（每个在 1 到 30 之间），然后对于每个读入的数，打印相应个数的连续星号。例如，如果程序读入数 7，则打印 \*\*\*\*\*。
- 3.12 一个邮购亭销售 5 种不同的产品，其零售价为：产品 1——\$2.98，产品 2——\$4.50，产品 3——\$9.98，产品 4——\$4.49，产品 5——\$6.87。编写一个 applet 程序，成对地读入一串数，如下所示：

1) 产品号

2) 每日售出数量

在程序中使用一个 switch 结构来确定每种产品的零售价, 计算并显示上周所有售出产品的总零售额, 使用一个文本字段获取产品号。

3.13 修改图 3.6 的程序, 使它只使用整数计算复利。(提示: 将所有金额表示为整数的美分。然后分别用除法和取模操作将其结果分为美元的部分和美分的部分, 在其中插入一个句点。)

3.14 假定  $i = 1$ 、 $j = 2$ 、 $k = 3$  和  $m = 2$ , 给出下列每条语句的打印结果。在每种情况中的括号都是必需的吗?

- a) `System.out.println( i -- 1 );`
- b) `System.out.println( j -- 3 );`
- c) `System.out.println( i >= 1 && j < 4 );`
- d) `System.out.println( m <= 99 & k < m );`
- e) `System.out.println( j >= i || k == m );`
- f) `System.out.println( k + m < j { 3 - j >= k );`
- g) `System.out.println( ! m );`
- h) `System.out.println( !( j - m ) );`
- i) `System.out.println( !( k > m ) );`

3.15 编写一个应用程序, 打印一个对应十进制数 1 到 256 的二进制、八进制、十六进制数的表格。如果不熟悉这些数值系统, 请先阅读附录 C “数值系统”。

3.16 使用无限数列来计算  $\pi$  的值:

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \dots$$

打印一个表格, 使用该数列的 1 项、2 项、3 项等近似表示  $\pi$  的值。在得到 3.14、3.141、3.1415、3.14159 之前使用了多少项?

3.17 (毕达哥拉斯三元组) 一个直角三角形的三边可为整数, 对应直角三角形各边的三个整数集合称为一个毕达哥拉斯三元组(勾股弦三元组)。这三条边必须满足两条直角边的平方之和等于斜边的平方。编写一个应用程序, 找出 500 以内所有对应 side1、side2 和 hypotenuse 的毕达哥拉斯三元组, 可以使用一个三重嵌套的 for 循环来尝试所有的可能性。这是一个“强制”计算的例子。在更高级的计算机科学的课程中, 有许多有趣的问题并没有替代强制计算的已知算法。

3.18 为了支付工资, 公司将雇员分为经理(固定的周薪)、小时工人(40 小时内有固定小时工资, 超额时间按 1.5 倍的小时工资计算)、佣金工人(每周 \$250 加上每周销售额的 2.5%) 或者计件工人(每生产一件产品获得一份固定数额的钱)。编写一个 applet 程序, 计算每个雇员的周薪。每种雇员都有其自身的工资支付代码: 经理有代码 1, 小时工人有代码 2, 佣金工人有代码 3, 计件工人有代码 4。使用一个 switch 结构来计算雇员基于其支付代码的工资。在 switch 结构内, 提示用户向该程序输入基于雇员支付代码的、计算每个雇员工资所需的相应数据。

3.19 (摩根定律) 在本章中, 我们讨论了逻辑运算符 `&&`、`&`、`||`、`!` 和 `!`。使用摩根定律有时能更方便地表达一个逻辑表达式。这些定律指出表达式 “`!(condition1 && condition2)`” 在逻辑上等价于表达式 “`!(condition1 || !condition2)`”。而且表达式 “`!(condition1 || condition2)`”



在逻辑上等价于表达式 “(!condition1 && !condition2)”。利用摩根定律写出下列表达式的等价表达式, 然后编写一个程序, 表明原来的表达式和新的表达式在每种情况下均相等。

- a) !( x < 5 ) && !( y >= 7 )
- b) !( a == b ) || !( g != 5 )
- c) !( ( x <= 8 ) && ( y > 4 ) )
- d) !( ( i > 4 ) || ( j <= 6 ) )

- 3.20 编写一个 applet 程序, 打印下面的菱形形状。可以在 paint 方法中使用一条输出语句来打印一个星号 (\*), 最大限度地利用循环 (使用嵌套的 for 语句) 结构, 并最大限度地减少输出语句。

```

      *
    ***
  *****
*****
*****
*****
  *****
    ***
      *

```

- 3.21 修改练习 3.21 中的程序, 读入 1 到 19 的奇数, 指定菱形的行数, 使程序显示不同大小的菱形。
- 3.22 break 语句和 continue 语句不支持结构化。实际上, break 语句和 continue 语句常常能由结构化语句所替代, 尽管这样可能会产生冗余。简要描述一下如何从一个循环中删除任何 break 语句, 并使用一些结构化等价语句来替代。(提示: break 语句从循环体内部跳出循环。另一种跳出循环的方法是使循环条件测试失败。考虑在循环条件测试中进行第二个测试, 表明“提前退出是由于一个 break 条件”。) 使用此方法从图 3.11 中的程序中删除 break 语句。
- 3.23 下面的程序段的功能是什么?

```

for ( i = 1; i <= 5; i ++ ) {

    for ( j = 1; j <= 3; j++ ) {

        for ( k = 1; k <= 4; k++ )
            System.out.print( '*' );
        System.out.println();
    }
    System.out.println();
}

```

- 3.24 简要描述如何从程序的循环中删除 continue 语句, 并使用一些结构化的等价语句来替换它。利用此方法从图 3.12 中的程序中删除 continue 语句。

# 第4章 方 法

## 教学目标

- 理解如何使用称为方法的小段程序来模块化地构造程序
- 介绍 Java API 中使用的数学方法
- 学会创建新方法
- 理解用于在方法间传递信息的机制
- 介绍利用随机数生成的模拟技术
- 理解标识符可见性如何局限在程序的特定区域内
- 理解如何编写和使用调用自己的方法

## 4.1 简介

大多数解决现实世界问题的程序都远远大于在前几章中提供的程序。经验表明,开发和维护一个大型程序的最佳办法,是使用那些比原始程序更易于管理的小段程序和模块 (module) 来构造程序。这一技术称为“分而治之,各个击破”。本章描述了 Java 语言的关键特征,例如设计、实现可以操作和维护大型程序的手段。

## 4.2 Java 中的程序模块

Java 中的模块称为方法 (method) 和类 (class)。Java 程序是通过将程序员编写的新方法与 Java API (也称为 Java 类库) 中“预打包” (pre-packaged) 的方法相组合,以及将程序员编写的新类与 Java 类库中“预打包”的类相组合而形成的。在本章中,我们将集中讨论方法,而在第 6 章中将详细讨论类。

Java API 提供了丰富的类和方法的集合,用于执行常见的数学计算、字符串操作、字符操作、输入/输出、错误检查以及其他有用的操作。由于这些方法提供了程序员所需的许多功能,因此能使程序员的工作更加简单。Java API 方法以 Java 2 软件开发工具箱 (JDK) 的形式提供。

### 编程技巧 4.1

熟悉 Java API 中的类和方法的集合,以及各种类库中丰富的类集合。

### 软件工程视点 4.1

避免反复重写程序。如果有可能,应使用 Java API 中的类和方法而不要编写新的类和方法,这样就缩短了程序的开发时间,并且减少了错误。

### 性能提示 4.1

不要重写已有的 Java API 类和方法以使其更有效率,一般情况下重写方法或类是不可能提高其效率的。

程序员可以通过编写方法来定义特定的任务,这些任务可以在一个程序的多处使用,有时也称为程序员定义的方法 (programmer-defined method)。定义方法的语句只需编写一次,而且这些语句对其他的方法是隐藏的。

一般通过方法调用 (method call) 来激活方法 (即执行其已定义的任务)。方法调用指定了方法名并提供了信息 (如参数), 这些是被调用方法完成任务所需要的。与此相类似的则是管理的层次模式。一位老板 (调用方法或调用者) 要求一位工人 (被调用方法) 完成一件任务, 在任务完成时返回结果 (即报告)。老板方法并不知道工人方法如何完成其定义好的任务, 工人方法可以调用其他工人方法, 老板方法将不会知道这一点。我们很快将看到这种“隐藏”实现细节的方法是如何推进良好的软件工程。图 4.1 显示了 main 方法以一种层次的方式同几个工人方法进行通信, 其中 worker1 是 worker4 和 worker5 的老板方法。方法间的各种关系可能不只是本图中显示的层次结构。

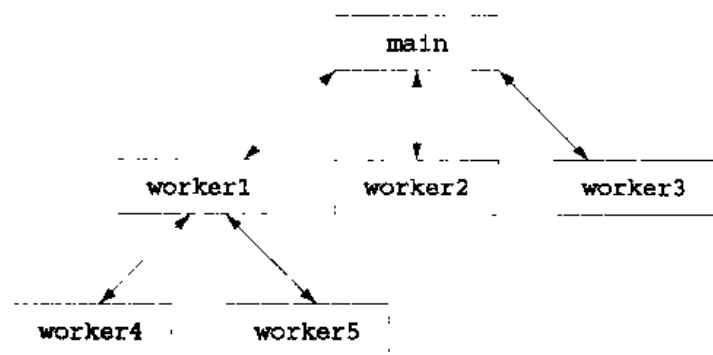


图 4.1 层次化的老板方法 / 工人方法的关系

### 4.3 Math 类的方法

Math 类的方法允许程序员完成一定的常见数学计算。我们在这里利用 Math 类方法来介绍方法的概念。在本书的稍后部分, 我们将讨论 Java API 中许多其他的方法。

一般是通过写出方法的名字, 后接一个左括号和参数 (或由逗号分开的参数表), 最后接一个右括号的形式来进行方法调用。例如, 可以使用如下语句计算 900.0 的平方根:

```
Math.sqrt ( 900.0 )
```

当执行这条语句时, 调用 static Math 方法 sqrt 来计算括号中 900.0 的平方根, 900.0 是 sqrt 的参数。上面语句的计算结果为 30.0。Math.sqrt 方法接收一个 double 类型的参数并返回 double 类型的结果。注意, 所有 Math 类的方法都是 static (静态) 类型的, 调用时必须在方法名前写上类名 Math 和句点 (.) 运算符。如果要输出计算结果, 则可以使用如下语句:

```
System.out.println( Math.sqrt( 900.0 ) );
```

在这条语句中, sqrt 返回的值是 println 方法的参数。

#### 软件工程视点 4.2

没有必要将 Math 类包含到一个程序中来使用其方法, Math 类是 java.lang 软件包的一部分, 将由编译器自动包含进来。

#### 常见编程错误 4.1

在调用 Math 类方法时忘记在方法名前写上类名 Math 和句点运算符 (.), 这将导致一个语法错误。

方法的参数可以是常量、变量或表达式。如果  $c1 = 13.0$ 、 $d = 3.0$ 、 $f = 4.0$ , 那么下列语句:

```
System.out.println ( Math.sqrt (c1 + d * f) );
```

将计算并打印 “ $13.0 + 3.0 \times 4.0 = 25.0$ ” 的平方根, 结果是 5.0。

图 4.2 总结了一些 Math 类的方法。在该图中, 变量  $x$  和  $y$  是 double 类型。Math 类也定义了两个常用的数学常量, 分别是 Math.PI 和 Math.E。其中常量 Math.PI (3.141 592 653 589 793 238 46) 是圆的周长对其直径的比率。常量 Math.E (2.718 281 828 459 045 235 4) 是自然对数的基数 (由 Math.log 方法计算得出)。

方法	描述	举例
abs(x)	x 的绝对值 (该方法还有 float、int 和 long 值的版本)	abs(23.7)的结果为 23.7 abs(0.0)的结果为 0.0 abs(-23.7)的结果为 23.7
ceil(x)	x 取整为不小于 x 的最小整数	ceil(9.2)的结果为 10.0 ceil(-9.8)的结果为 -9.0
cos(x)	x 的三角余弦 (x 值为弧度)	cos(0.0)的结果为 1.0
exp(x)	指数方法 $e^x$	exp(1.0)的结果为 2.718 28 exp(2.0)的结果为 7.389 06
floor(x)	x 取整为不大于 x 的最大整数	floor(9.2)的结果为 9.0 floor(-9.8)的结果为 -10.0
log(x)	x 的自然对数 (基数为 e)	log(2.718 282)的结果为 1.0 log(7.389 056)的结果为 2.0
max(x, y)	取 x 和 y 中较大的值 (该方法还有 float、int 和 long 值的版本)	max(2.3, 12.7)的结果为 12.7 max(-2.3, -12.7)的结果为 -2.3
min(x, y)	取 x 和 y 中较小的值 (该方法还有 float、int 和 long 值的版本)	min(2.3, 12.7)的结果为 2.3 min(-2.3, -12.7)的结果为 -12.7
pow(x, y)	x 的 y 次幂 ( $x^y$ )	pow(2, 7)的结果为 128.0 pow(9, .5)的结果为 3.0
sin(x)	x 的三角正弦 (x 值为弧度)	sin(0.0)的结果为 0.0
sqrt(x)	x 的平方根	sqrt(900.0)的结果为 30.0 sqrt(9.0)的结果为 3.0
tan(x)	x 的三角正切 (x 值为弧度)	tan(0.0)的结果为 0.0

图 4.2 常用的 Math 类的方法

## 4.4 方法

方法允许程序员模块化一个程序。所有在方法定义中声明的变量均为局部变量——它们仅在定义的方法中可见。大多数方法都有一个形式参数表, 从而提供了方法间交换信息的手段。方法的参数也是局部变量。

使用方法来模块化程序有几个目的。首先, “分而治之, 各个击破” 的方法使得程序的开发更好管理。另一个目的是软件的可重用性, 可以将现有的方法作为构件块来创建新的程序。只要使用良好的方法命名和定义, 程序便可以通过标准方法进行创建, 而不必使用用户化的代码。第三个目的是避免程序中的重复代码, 将代码作为一个方法打包起来, 这样在程序的其他地方执行这些代码时, 只需简单地调用该方法即可。

**软件工程视点 4.3**

应限制每个方法来完成一个单一的、定义良好的任务，方法名应能有效地表达该任务。这将提高软件的可重用性。

**软件工程视点 4.4**

如果不能选择一个简洁的名字来表达方法的作用，这可能说明该方法要完成太多种类的任务。通常，最好将这样的方法分成几个较小的方法。

## 4.5 方法定义

我们提供的每个程序都包括一个类定义，每个类定义中至少含有一个称为Java API方法的方法定义。我们现在介绍如何编写定制的方法。

考虑在 applet 程序中使用一个方法 square（在 applet 的 paint 方法中激活），以计算整数 1 到 10 的平方（如图 4.3 所示）。

```
1 // Fig. 4.3: SquareInt.java
2 // A programmer-defined square method
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class SquareInt extends Applet {
7     // output the squared values of 1 through 10
8     public void paint( Graphics g )
9     {
10         int xPosition = 25;
11
12         for ( int x = 1; x <= 10; x++ ) {
13             g.drawString( String.valueOf( square( x ) ),
14                           xPosition, 25);
15             xPosition += 20;
16         }
17     }
18
19     // square method definition
20     public int square(int y)
21     {
22         return y * y;
23     }
24 }
```

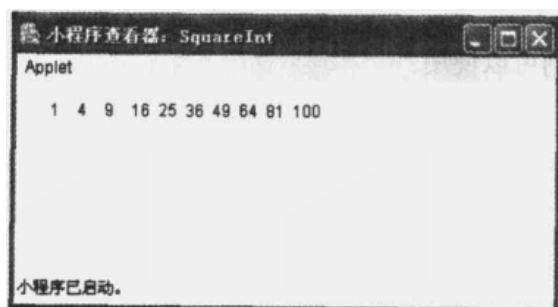


图 4.3 使用程序员定义的方法

#### 编程技巧 4.2

在方法定义之间放一个空行来分隔方法，从而增强程序的可读性

在 paint 的第 13 行上，使用下面的形式来激活或调用 square 方法：

```
square ( x )
```

square 方法在参数 y 中接收 x 值的一份副本。接着，square 方法计算  $y*y$ ，结果将传递回 paint 方法中激活 square 的位置，并同时显示结果。这一过程使用 for 循环结构重复了 10 次。

square 方法的定义（第 20 行）显示 square 接收一个整型参数 y，方法名前的关键字 int 表明 square 将返回一个整数结果，square 中的 return 语句将计算的结果传递回调用的方法。注意，整个方法定义是位于 SquareInt 类的花括号中。Java 的所有方法必须在一个类的定义中进行定义。

#### 常见编程错误 4.2

在类的定义之外定义方法会导致语法错误。

一个方法定义的通常格式为：

```
返回值类型 方法名 ( 参数表 )  
:  
    声明和语句
```

方法名是任意合法的标识符，返回值类型是该方法返回其调用者的结果数据类型，返回值类型为 void 表明该方法没有返回值。方法至少可以返回一个值。

参数表是一个由逗号分开的，包含调用时方法接收的各个参数声明的列表。如果一个方法并不接收任何值，则参数表为空（即一对空括号跟在方法名后）。每个参数的类型都必须在方法的参数表中显式地列出。例如，一个 double 类型的参数可以接收 7.35、22 或 -0.035 46 这样的值，而不能接收 “hello” 这样的值（因为字符串不能赋值给 double 变量）。如果方法不接收任何值，则它的参数表是空的（即方法名后仅跟一对括号）。方法参数表中的每个参数都必须声明其数据类型，否则将产生语法错误。

在方法定义行（称为方法首部）之后，花括号中的声明和语句组成了方法体，方法体也称为一个块，一个块就是包括声明的复合语句。变量可以在任意块中声明，块也可以嵌套。一个方法不能在另一个方法中定义。

有三种方式将控制返回到激活方法的语句上。如果该方法不返回一个值，则程序控制在执行到方法的右花括号时只是简单地返回，或者执行下列语句：

```
return;
```

如果方法要返回一个值，则可以使用：

```
return expression;
```

该语句将表达式的值返回给调用者。当执行一个返回语句时，控制将立即返回到激活方法的位置。

注意，图 4.3 的例子实际上包含了两个方法定义—— paint（第 8 行）和 square（第 20 行）。请记住，paint 方法将自动调用以使 applet 显示信息。在这个例子中，paint 方法重复激活 square 方法来完成一次计算，接着输出结果。允许一个类定义中的方法激活同一个类定义中的其他方法（第 6 章将讨论一个例外情况）。

**常见编程错误 4.3**

在方法定义中省略返回值类型将会导致语法错误。

**常见编程错误 4.4**

忘记从一个要返回值的方法中返回值将产生语法错误。如果指定了一个非 void 的返回值类型，则该方法必须包含一条 return 语句。

**常见编程错误 4.5**

从一个返回类型已声明为 void 的方法返回一个值会导致语法错误。

**常见编程错误 4.6**

把相同类型的方法参数声明为“float x, y”而非“float x, float y”，则参数声明“float x, y”实际上会报告编译错误，因为参数表中的每个参数都需要有类型。

**常见编程错误 4.7**

在方法定义参数表的右括号后放一个分号将产生语法错误。

**常见编程错误 4.8**

在方法体中定义参数将产生语法错误。

**常见编程错误 4.9**

将方法的参数作为一个局部变量而再次定义将产生语法错误。

**常见编程错误 4.10**

在一个方法中定义另一个方法将产生语法错误。

**编程技巧 4.3**

尽管并非不正确，建议不要为传递给方法的实参以及在方法定义中相应的形参取相同的名字，这有助于避免混淆。

**编程技巧 4.4**

选择有意义的方法名和有意义的参数名会使程序的可读性更好，并有助于避免过多地使用注释。

**软件工程视点 4.5**

一个方法通常应当不长过一页，最好不要长过半页。不管一个方法有多长，它都应当很好地执行一个任务。规模小的方法可以提高软件的可重用性。

**软件工程视点 4.6**

程序应当成为小方法的集合。这使得程序易于编写、调试、维护和修改。

**软件工程视点 4.7**

一个需要大量参数的方法可能是要完成的任务太多了。应考虑将该方法分为较小的方法，然后完成分解后的各个小任务。如果有可能，方法的首部应放置在一行上。

**软件工程视点 4.8**

方法首部和方法调用在实参与形参的数量、类型、顺序以及返回值的类型上都应相同。

第二个例子使用了一个由用户定义的、称为 maximum 的方法来判断并返回三个整数中最大的一个（如图 4.4 所示）。

三个整数由用户输入到文本字段中。当用户在一个文本字段中按下回车键时，就激活 action 方

法, 读取文本字段中的字符串并转换成整数, 这三个整数将传递给确定最大值的 `maximum` 方法。由 `maximum` 方法计算出最大值, 并由 `return` 语句返回给 `action` 方法。返回值将赋给变量 `max`, 并随后在 `result` 文本字段中显示出来。

```
1 // Fig. 4.4: Maximum.java
2 // Finding the maximum of three integers
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class Maximum extends Applet {
7     Label label1, label2, label3, resultLabel;
8     TextField number1, number2, number3, result;
9     int num1, num2, num3, max;
10
11     // set up labels and text fields
12     public void init()
13     {
14         label1 = new Label( "Enter first integer:" );
15         number1 = new TextField( "0", 10 );
16         label2 = new Label( "Enter second integer:" );
17         number2 = new TextField( "0", 10 );
18         label3 = new Label( "Enter third integer:" );
19         number3 = new TextField( "0", 10 );
20         resultLabel = new Label( "Maximum value is:" );
21         result = new TextField( "0", 10 );
22         result.setEditable( false );
23
24         add( label1 );
25         add( number1 );
26         add( label2 );
27         add( number2 );
28         add( label3 );
29         add( number3 );
30         add( resultLabel );
31         add( result );
32     }
33
34     // maximum method definition
35     public int maximum( int x, int y, int z )
36     {
37         return Math.max( x, Math.max( y, z ) );
38     }
39
40     // get the integers and call the maximum method
41     public boolean action( Event e, Object o )
42     {
43         num1 = Integer.parseInt( number1.getText() );
44         num2 = Integer.parseInt( number2.getText() );
45         num3 = Integer.parseInt( number3.getText() );
46         max = maximum( num1, num2, num3 );
47         result.setText( Integer.toString( max ) );
48         return true;
49     }
50 }
```



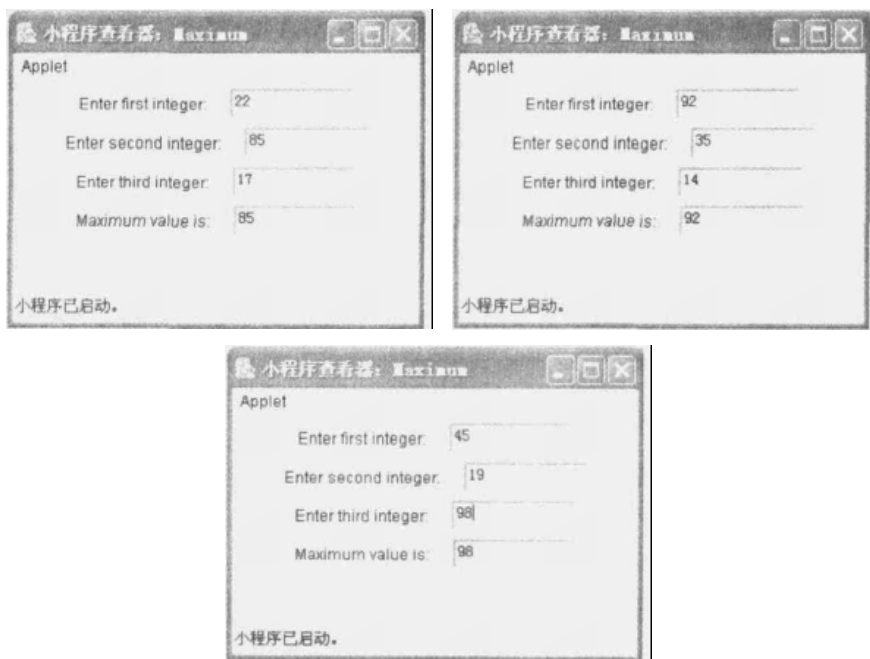


图 4.4 程序员定义的 maximum 方法

注意方法 `maximum` (第 35 行) 的实现。第一行表明: 该方法返回一个整数, 该方法的名字为 `maximum`, 并且该方法接收三个整型参数 (`x`、`y` 和 `z`) 来完成其任务。方法的程序体包含下列语句:

```
return Math.max ( x, Math.max ( y, z ) );
```

这条语句两次调用 `Math.max` 方法, 以返回三个整数中最大的一个。首先, 激活方法 `Math.max`, 判断变量 `y` 和 `z` 这两个值中较大的一个。接着, 将变量 `x` 的值和第一次对 `Math.max` 调用的结果传递给 `Math.max` 方法。最后, 第二次调用 `Math.max`, 返回到激活 `maximum` 的语句 (即此程序中的 `action` 方法)。

## 4.6 参数类型提升

方法定义的另一个重要特征是参数的强制类型转换 (coercion of argument), 即强制参数成为传递到方法中的合适类型。例如, 可以使用一个整型 (`int`) 参数调用 `Math` 类的方法 `sqrt`。即使该方法在 `Math` 类中定义为接收一个 `double` 参数, `sqrt` 方法仍能正确地工作。

下列语句:

```
System.out.println ( Math.sqrt (4) );
```

用来正确地计算 `Math.sqrt (4)`, 并打印出值 2。在将值传递到 `Math.sqrt` 之前, 方法定义的参数表可以将整数值 4 转换成 `double` 值 4.0。在许多情况下, 需要在方法调用前将与方法定义中的参数类型不一致的参数值转换成合适的类型。在某些情况下, 如果未遵从 Java 的提升规则 (promotion rule), 那么这些转换可能导致编译错误。提升规则定义了在不丢失数据的前提下, 如何将一个类型转换成其他类型。在前面的 `Math.sqrt` 例子中, 一个 `int` 类型在不改变其值的前提下自动转换成了 `double` 类型, 但是在将一个 `double` 类型转换成 `int` 类型时却去除了 `double` 值的小数部分。将大整数类型转换成小整数类型 (例如 `long` 到 `int`) 也可能导致数值的改变。

提升规则适用于包含两个或更多数据类型的表达式, 这样的表达式也称为混合类型表达式。混合类型表达式中每个值的类型都将提升为表达式中的“最高”类型 (实际上创建每个值的临时版本

并用于表达式，即原始值保持不变）。图 4.5 中列出了基本数据类型以及允许自动提升的类型。

类型	允许的提升
double	无
float	double
long	float 或 double
int	long、float 或 double
char	int、long、float 或 double
short	int、long、float 或 double
byte	short、int、long、float 或 double
boolean	无

图 4.5 基本数据类型可以进行的提升

将值转换成较低的类型可能导致不正确的值。因此，如果转换时可能导致信息丢失，则 Java 编译器要求程序员使用类型转换运算符来强制发生转换。例如，为了使用 double 变量 y 调用带有一个整型参数的 square 方法（如图 4.3 所示），我们使用方法调用“square((int)y)”，这样就显式地在 square 方法中将 y 的值转换成了一个整数。因此，如果 y 的值为 4.5，则 square 方法将返回 16 而非 20.25。

#### 常见编程错误 4.11

将一个基本数据类型的值转换成另一个基本数据类型，而后一个类型又不在允许提升的原始类型的列表中，那么将会改变数据的值。

## 4.7 Java API 软件包

正如我们所看到的，Java 包含许多称为类的预定义程序段，它们在磁盘上按目录分组，从而形成相关类的分类，通常称之为软件包。综合来说，这些软件包称为 Java 应用程序编程接口（Java API）或 Java 类库（Java class library）。

我们在这里使用 import 语句来加载用以编译 Java 程序的类。例如，为了告诉编译器从 java.applet 软件包中加载 Applet 类，可以使用下列语句：

```
import java.applet.Applet;
```

Java 的优势之一就是 Java API 软件包中含有大量的类，Java API 可由程序重用而防止“反复重写”一段程序，我们会在本书中使用到这些类中的大部分。图 4.6 中按字母顺序列出了 Java API 软件包，并对每个软件包进行了说明。在这个表中，介绍了大量 Java API 中可重用的组件。在学习 Java 时，读者需要首先阅读 Java API 说明文档中关于软件包和类的主题（可以参考 WWW 站点 [java.sun.com/j2se/1.3/docs/api](http://java.sun.com/j2se/1.3/docs/api)）。

Java API 软件包	描述
java.applet	Java applet 软件包 这个软件包中包含了 Applet 类和几个接口，可以用来创建 applet，使用浏览器同 applet 进行交互，并播放音频剪辑。在 Java 2 中，使用 javax.swing.JApplet 类来定义在 Swing GUI 组件中使用的 applet。
java.awt	Java 抽象窗口化工具箱软件包（Java Abstract Windowing Toolkit Package） 这个软件包中包含用来在 Java 1.0 和 Java 1.1 中创建和维护图形用户界面的所有类和接口。在 Java 2 中，这些类仍可使用，不过更常使用 Swing GUI 组件中的 javax.swing 软件包。
java.awt.event	Java 抽象窗口化工具箱事件包（Java Abstract Windowing Toolkit Event Package） 这个软件包中包含一些在 java.awt 和 java.swing 软件包中允许 GUI 组件进行事件处理的类和接口。
java.io	Java 输入/输出软件包 这个软件包中包含了使程序能够输入和输出数据的类（详细内容请参见第 15 章“文件和流”）。

(续表)

Java API 软件包	描述
java.lang	Java 语言包 这个软件包自动由编译器引入到所有程序, 其中包含了基本的类和接口, 可用于许多 Java 程序
java.net	Java 网络包 这个软件包中包含了使程序通过网络进行通信的类(参见第 16 章“网络”)
java.text	Java 文本包 这个软件包中包含了可以使 Java 操纵成员(数据、字符和字符串)的类和接口, 提供了许多 Java 的国际化功能, 即允许将程序定制为具有某个特定地区的特性(例如, 可以使 applet 程序基于用户所在的国家而使用不同的语言显示字符等)
java.util	Java 工具包 这个软件包中包含了实用工具类和接口, 例如日期和时间操作, 各种随机数处理手段(Random), 存储并处理大量的数据, 将字符串分为较小的称为标记的片段(StringTokenizer), 以及其他的方法(详细内容请参见第 17 章、第 18 章)
javax.swing	Java Swing GUI 组件包 这个软件包中包含了为可移植 GUI 提供支持的 Java Swing GUI 组件所使用的类和接口
javax.swing.event	Java Swing 事件包 这个软件包中包含了在 javax.swing 软件包中允许 GUI 组件进行事件处理的类和接口

图 4.6 Java API 软件包

## 4.8 生成随机数

在本节和下一节中, 我们将开发一个结构良好的游戏程序, 其中包含多种方法。该程序使用了我们研究过的大多数控制结构。

赌场上的某种东西吸引着各种各样的人, 这就是机会元素(element of chance), 即将一袋钱变为一座金山的概率。机会元素可由 Math 类的 random 方法引入。

考虑下面的语句:

```
double randomValue = Math.random ( );
```

random 方法产生一个 0.0 到 1.0 (不包含) 的 double 值。如果 random 的确产生随机值, 那么每个从 0.0 到 1.0 (不包含) 的值在每次调用 random 方法时都有均等的选中机会(或概率)。

直接由 random 产生的值的范围常常与一个特定应用所需要的不同。例如, 一个模拟抛硬币的程序也许只需要 0 来表示“正面”和 1 来表示“反面”。而一个模拟掷六面骰子的程序则需要从 1 到 6 的随机整数。一个随机预测下一个飞过视频游戏地平线的飞船类型(四种可能之一)的程序, 则将需要使用从 1 到 4 的随机整数。

为了更好地展示 random 方法, 让我们开发一个程序来模拟掷骰子 20 次并打印每次的结果值。使用乘法运算符(\*)与 random 方法一起来产生从 0 到 5 的整数, 如下所示:

```
(int)(Math.random ( ) * 6 )
```

这种方式称为比例缩放, 数字 6 称为缩放因子。我们随后通过向前面的结果加 1 来提升数字范围。图 4.7 表明结果范围为 1 到 6。整数类型强制转换运算符用来删除前面表达式产生的每个值的浮点部分(小数点后面的部分)。

为了显示这些数字的产生大致是机会均等的, 让我们使用图 4.8 的程序模拟掷一个骰子 6 000 次。从 1 到 6 的每个整数应当各出现约 1 000 次。

正如程序输出所表明的, 通过比例缩放和移位, 我们可以利用 random 方法来实际模拟一个六面骰子的滚动。注意, 在 switch 结构中没有提供 default 条件。在第 5 章研究了数组后, 我们将指明如何使用一行语句来替换整个 switch 语句。运行几次程序并观察其结果, 注意, 每次程序执行都将产生一个不同的随机数列。

直接由 random 方法产生的值通常都在下面的范围中:

$$0.0 \leq \text{Math.random}() < 1.0$$

```

1 // Fig. 4.7: RandomInt.java
2 // Shifted, scaled random integers
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class RandomInt extends Applet {
7     public void paint( Graphics g )
8     {
9         int xPosition = 25;
10        int yPosition = 25;
11        int value;
12
13        for ( int i = 1; i <= 20; i++ ) {
14            value = 1 + (int) ( Math.random() * 6 );
15            g.drawString( Integer.toString( value ),
16                        xPosition, yPosition );
17
18            if ( i % 5 != 0 )
19                xPosition += 40;
20            else {
21                xPosition = 25;
22                yPosition += 15;
23            }
24        }
25    }
26 }

```

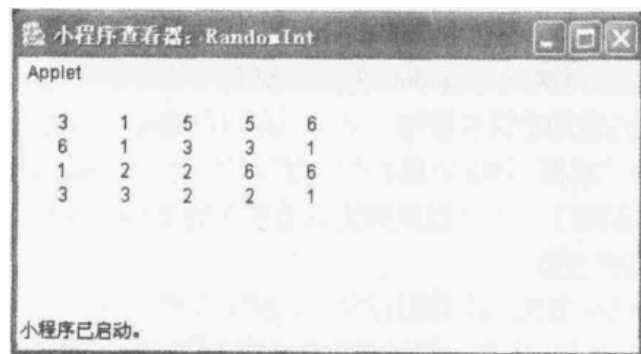


图 4.7 比例缩放和进行移位所产生的随机整数

```

1 // Fig. 4.8: RollDie.java
2 // Roll a six-sided die 6000 times
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class RollDie extends Applet {

```

```
7      int frequency1 = 0, frequency2 = 0,
8          frequency3 = 0, frequency4 = 0,
9          frequency5 = 0, frequency6 = 0;
10
11      // summarize results
12      public void start()
13      {
14          for ( int roll = 1; roll <= 6000; roll++ ) {
15              int face = 1 + (int) ( Math.random() * 6 );
16
17              switch ( face ) {
18                  case 1:
19                      ++frequency1;
20                      break;
21                  case 2:
22                      ++frequency2;
23                      break;
24                  case 3:
25                      ++frequency3;
26                      break;
27                  case 4:
28                      ++frequency4;
29                      break;
30                  case 5:
31                      ++frequency5;
32                      break;
33                  case 6:
34                      ++frequency6;
35                      break;
36              }
37          }
38      }
39
40      // display results
41      public void paint( Graphics g )
42      {
43          g.drawString( "Face", 25, 25 );
44          g.drawString( "Frequency", 100, 25 );
45          g.drawString( "1", 25, 40 );
46          g.drawString( Integer.toString( frequency1 ), 100, 40 );
47          g.drawString( "2", 25, 55 );
48          g.drawString( Integer.toString( frequency2 ), 100, 55 );
49          g.drawString( "3", 25, 70 );
50          g.drawString( Integer.toString( frequency3 ), 100, 70 );
51          g.drawString( "4", 25, 85 );
52          g.drawString( Integer.toString( frequency4 ), 100, 85 );
53          g.drawString( "5", 25, 100 );
54          g.drawString( Integer.toString( frequency5 ),
55                      100, 100 );
56          g.drawString( "6", 25, 115 );
57          g.drawString( Integer.toString( frequency6 ),
58                      100, 115 );
59      }
60  }
```

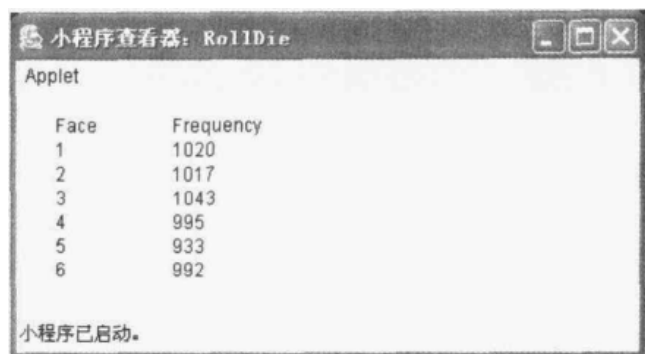


图 4.8 掷骰子 6 000 次

前面我们曾说明了如何使用一条语句来模拟掷六面骰子：

```
int face = 1 + ( int ) ( Math.random ( ) * 6 );
```

该语句总是将一个整数（随机）赋给变量 `face`，其范围为  $1 \leq \text{face} \leq 6$ 。注意，此范围的宽度（即此范围中连续整数的个数）是 6，并且范围的开始值是 1。参考前面的语句，我们看到此范围的宽度是由与乘法运算符一起比例缩放 `random` 的数字（即 6）决定的，而此范围的开始数字等于加到 “(int) (Math.random() \* 6)” 上的数字（即 1）。我们可以使用下面的语句来产生这一结果：

```
n = a + ( int ) ( Math.random ( ) * b );
```

其中 `a` 是位移值（它等于期望的连续整数范围的第一个数值），而 `b` 是比例因子（它等于期望的连续整数的宽度）。在练习中，我们将看到有可能从值的集合中选择随机整数而并非从连续整数的范围中选择随机整数。

## 4.9 案例：一个机会游戏

一个最常见的机会游戏是掷骰子游戏，游戏的规则如下：

一名玩家掷两个骰子。每个骰子有六个面。这些面包含 1、2、3、4、5 和 6。在骰子停止滚动后，算出两个朝上面的点数和。如果头一次掷出的和为 7 或 11，则玩家就赢了；如果头一次掷出的和为 2、3 或 12，则玩家就输了（即“庄家”赢了）。如果头一个和为 4、5、6、8、9 或 10，则此和就变成玩家的“点”数。为了赢局，必须接着掷骰子，直到“得到想要的点数”。如果在投出自己的点数之前投出了 7 这个点数，则玩家就输了。

图 4.9 的程序模拟了掷骰子的游戏。

注意，玩家头一次掷骰子时必须掷两个骰子，以后也都是如此。我们定义了一个 `rollDice` 方法（第 106 行）来滚动骰子并计算及显示它们的和。`rollDice` 方法定义了一次，但它在程序中的两处被调用（第 53 行和第 75 行）。有趣的是，`rollDice` 没有参数，于是我们给出了一个空参数表。`rollDice` 方法返回两个骰子的和，因此要在方法首部中指明 `int` 返回类型。

```
1 // Fig. 4.9: Craps.java
2 // Craps
3 import java.awt.*;
4 import java.applet.Applet;
```

```

5
6 public class Craps extends Applet {
7     // constant variables for status of game
8     final int WON = 0, LOST = 1, CONTINUE = 2;
9
10    // other variables used in program
11    boolean firstRoll = true;    // true if first roll
12    int dieSum = 0;              // sum of the dice
13    int myPoint = 0;             // point if no win/loss on first roll
14    int gameStatus = CONTINUE;   // WON, LOST, CONTINUE
15
16    // graphical user interface components
17    Label die1Label, die2Label, sumLabel, pointLabel;
18    TextField firstDie, secondDie, sum, point;
19    Button roll;
20
21    // setup graphical user interface components
22    public void init()
23    {
24        die1Label = new Label( "Die 1" );
25        firstDie = new TextField( 10 );
26        firstDie.setEditable( false );
27        die2Label = new Label( "Die 2" );
28        secondDie = new TextField( 10 );
29        secondDie.setEditable( false );
30        sumLabel = new Label( "Sum is" );
31        sum = new TextField( 10 );
32        sum.setEditable( false );
33        roll = new Button( "Roll Dice" );
34        pointLabel = new Label( "Point is" );
35        point = new TextField( 10 );
36        point.setEditable( false );
37
38        add( die1Label );
39        add( firstDie );
40        add( die2Label );
41        add( secondDie );
42        add( sumLabel );
43        add( sum );
44        add( pointLabel );
45        add( point );
46        add( roll );
47    }
48
49    // process one roll of the dice
50    public void play()
51    {
52        if ( firstRoll ) {                // first roll of the dice
53            dieSum = rollDice();
54
55            switch( dieSum ) {
56                case 7: case 11:           // win on first roll
57                    gameStatus = WON;
58                    point.setText( "" );    // clear point text field
59                    firstRoll = true;       // allow new game to start

```

```

60         break;
61         case 2: case 3: case 12:           // lose on first roll
62             gameStatus = LOST;
63             point.setText( "" );           // clear point text field
64             firstRoll = true;              // allow new game to start
65             break;
66         default:                           // remember point
67             gameStatus = CONTINUE;
68             myPoint = dieSum;
69             point.setText( Integer.toString( myPoint ) );
70             firstRoll = false;
71             break;
72     }
73 }
74 else {
75     dieSum = rollDice();
76
77     if ( dieSum == myPoint )                // win by making point
78         gameStatus = WON;
79     else
80         if ( dieSum == 7 )                  // lose by rolling 7
81             gameStatus = LOST;
82 }
83
84 if ( gameStatus == CONTINUE )
85     showStatus( "Roll again." );
86 else {
87     if ( gameStatus == WON )
88         showStatus( "Player wins. " +
89                     "Click Roll Dice to play again." );
90     else
91         showStatus( "Player loses. " +
92                     "Click Roll Dice to play again." );
93
94     firstRoll = true;
95 }
96 }
97
98 // call method play when button is clicked
99 public boolean action( Event e, Object o )
100 {
101     play();
102     return true;
103 }
104
105 // roll the dice
106 int rollDice()
107 {
108     int die1, die2, workSum;
109
110     die1 = 1 + (int) ( Math.random() * 6 );
111     die2 = 1 + (int) ( Math.random() * 6 );
112     workSum = die1 + die2;
113
114     firstDie.setText( Integer.toString( die1 ) );

```



```

115         secondDie.setText( Integer.toString( die2 ) );
116         sum.setText( Integer.toString( workSum ) );
117
118         return workSum;
119     }
120 }

```

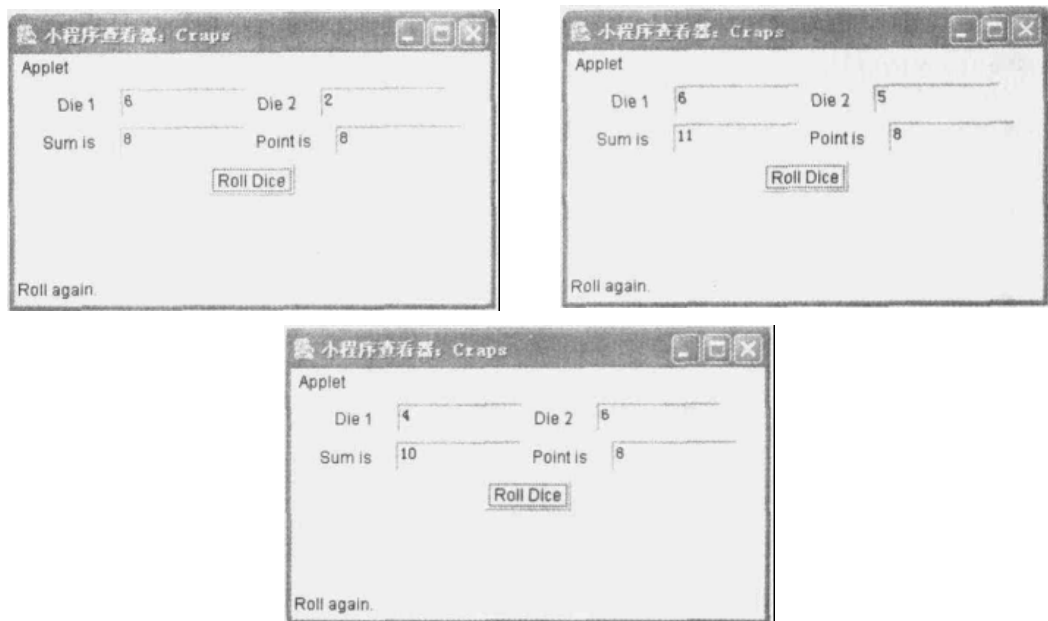


图 4.9 模拟掷骰子游戏

程序中合理地引入了游戏。玩家在第一次掷骰子时就可能赢或输，也可能在之后的任何一次掷骰子中赢或输，变量 `gameStatus` 用于跟踪这一情况，将该变量声明为 `int` 类型。程序的第 8 行：

```
final int WON = 0, LOST = 1, CONTINUE = 2;
```

创建了定义一个骰子三种状态的变量——赢局、输局或继续掷骰子。关键字 `final` 在声明的开始表明这些都是常量变量。常量变量在声明时必须进行初始化，在程序运行时则不能修改。常量变量常常称为命名常量或只读变量。`final` 变量的名字一般要大写，这样它们在程序中就会十分醒目，我们将在第 5 章和第 6 章中详细讨论 `final` 关键字。

#### 常见编程错误 4.12

在定义一个 `final` 变量之后，试图将另一个值赋给该变量将产生语法错误。

#### 编程技巧 4.5

在 `final` 变量的名字中只使用大写字母，这样这些常量在程序中就会很醒目。

#### 编程技巧 4.6

使用 `final` 变量而非整数常量（如 2），可以使程序的可读性更高。

用户按下 Roll Dice 按钮来掷骰子，这样就激活了 applet 的 `action` 方法，随后又激活了 `play` 方法。`play` 方法检查 `boolean` 变量 `firstRoll`，以判断它是 `true` 或 `false`（如果为 `true`，则这是游戏中第一次掷骰子）。在头一次掷骰子之后，如果游戏已经赢了或输了，那么程序就前进到第 87 行的 `if/else` 结构，如果 `gameStatus` 等于 `WON`，则在状态栏中显示 “Player wins.Click Roll Dice to Play again”（玩家赢

了。单击 Roll Dice 按钮再玩一次)。如果 gameStatus 等于 LOST, 则显示 “Player Loses. Click Roll Dice to Play again” (玩家输了。单击 Roll Dice 按钮再玩一次)。在第一次掷骰子之后, 如果游戏还没有结束, 则将 sum 存储在 myPoint 中, 并在 point 文本字段中显示。每次单击 Roll Dice 按钮后, 激活 play 方法并且调用 rollDice 方法以产生一个新 sum。如果 sum 正好为 myPoint, 则将 gameStatus 置为 WON, 执行第 87 行的 if/else 结构, 游戏结束。这时, 再次单击 RollDice 按钮则又启动了一次新游戏。在该程序中, 对于每次掷骰子, 4 个文本字段都使用新的骰子值及总和值进行更新, 而且每次新游戏开始时都将更新 point 文本字段。

请注意我们讨论过的各种程序控制机制的应用程序。掷骰子程序使用 4 个方法, 包括 init、action、play 和 rollDice, 以及 switch、if/else 和嵌套 if 结构。在练习中, 我们要考察掷骰子游戏的各种有趣特性。

## 4.10 自动变量

在第 1 章到第 3 章中, 我们使用了变量名的标识符。变量的属性包括名字、类型、存储大小和值。在本章中, 我们也使用标识符作为用户自定义方法和类的名字。实际上, 程序中每个标识符都有其他的属性, 包括生存期 (duration) 和作用域 (scope)。

标识符的生存期 (也称生命周期) 决定了它在内存中的存在周期。一些标识符短暂存在, 一些则将其反复创建和删除, 另一些则在程序的整个执行过程中都存在。

一个标识符的作用域是其在程序中可以引用的范围。一些标识符可以在整个程序中引用, 另一些只能从程序的有限位置引用。本节将介绍标识符的生存期, 下一节将介绍标识符的作用域。

一个方法中表示局部变量的标识符 (即方法体中声明的参数和变量) 拥有自动生存期 (automatic duration)。自动生存期变量在进入声明它们的块时创建, 当该块活动时存在, 当退出该块时将其删除, 我们称自动生存期变量为自动变量。

### 性能提示 4.1

自动生存期是一种节约内存的手段, 因为自动生存期变量在进入声明它们的块时创建, 在从块退出时就会将其删除。

如果程序员没有提供初始值, 则由编译器自动初始化类的实例变量。除了将 boolean 变量初始化为 false 之外, 基本数据类型的变量均初始化为 0, 引用将初始化为 null (空)。与类的实例变量不同, 自动变量在使用之前必须由程序员初始化。

### 测试与调试提示 4.1

自动变量在一个方法使用它们之前必须初始化, 否则编译器会发出一条错误消息。

Java 也有静态生存期 (static duration) 的标识符。静态生存期的变量和方法从它们定义的类加载至内存中开始运行时起, 直到程序终止时一直存在。对于静态生存期变量, 当把它们的类加载至内存时, 存储空间将一次性分配和初始化。对于静态生存期方法, 当它们的类加载至内存中时, 方法的名字就已经存在了。即使将它们的类加载至内存时静态生存期变量和方法名已经存在, 也并不表示这些标识符在整个程序中都可使用。生存期和作用域 (一个变量可以使用的范围) 是两个各自独立的概念, 这一点将在下一节中阐明。

### 软件工程视点 4.9

自动生存期还是最小权限原则的一个例子。这个规则说明系统的每一个组件都应当拥有充分权限, 以完

成设计任务, 不过没有附加的权限, 这个约束有助于避免程序中发生的意外或严重错误。当变量已经不再需要时, 我们为什么还让它们存储在内存中而且还可以访问呢?”

## 4.11 作用域规则

标识符的作用域是指在程序中的某一部分可以引用标识符。例如, 当我们在一个块中声明一个局部变量时, 该变量只能在这个块中或嵌套在其中的块内引用。一个标识符的作用域有类作用域 (class scope) 和块作用域 (block scope) 之分。在 break 和 continue 语句中还有一种用于标签的特殊作用域, 标签只在紧跟其后的循环结构体中可用。

一个类的方法和实例变量有类作用域, 类作用域从类定义的左花括号 ( { ) 开始, 并在类定义的右花括号 ( } ) 处结束。类作用域使一个类的方法能直接激活在同一个类中定义的方法或继承的方法 (如从 Applet 类继承的 applet 程序中的方法), 并且使之能直接访问在类中定义的所有实例变量 (在第6章中, 我们将看到 static 方法是此规则的一个例外)。在这种情况下, 一个类的所有实例变量和方法对于定义它们的类中的方法都是全局的, 即方法可以直接修改实例变量并激活类的其他方法。

在块内声明的标识符具有块作用域。块作用域在标识符声明的地方开始, 并且在块的右花括号处结束。在方法开始处声明的局部变量的块作用域同方法的参数一样, 参数也是方法的局部变量。任何块都可以包含变量声明, 当块有嵌套时, 如果外层块的标识符与内层块的标识符同名, 编译器会生成语法错误, 表明该变量已经定义过了。如果方法中的局部变量与实例变量同名, 则实例变量将“隐藏”起来直到块结束执行为止。在第6章中, 我们将讨论如何访问这类“隐藏”的实例变量。

### 常见编程错误 4.13

在内层块中使用与外层块中的标识符同名的标识符, 会使编译器产生语法错误。

### 编程技巧 4.7

应尽量避免局部变量隐藏实例变量的情况, 这可以通过避免在程序中使用重复名的办法来解决。

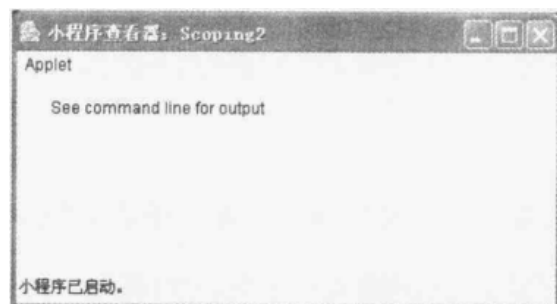
图 4.10 中的程序展示了实例变量和自动局部变量的作用域。注意, 将程序的输出送至命令行上 (如果正在使用一个浏览器, 则可以输出至 Java 控制台上)。

```
1 // Fig. 4.10: Scoping.java
2 // A scoping example
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class Scoping2 extends Applet {
7     int x = 1;           // instance variable
8
9     public void paint( Graphics g )
10    {
11        g.drawString( "See command line for output", 25, 25 );
12
13        int x = 5;        // local variable to paint
14
15        System.out.println( "local x in paint is " + x );
16
17        a();              // a has automatic local x
```

```

18         b();                // b uses instance variable x
19         a();                // a reinitializes automatic local x
20         b();                // instance variable x retains its value
21
22         System.out.println( " \nlocal x in paint is " + x );
23     }
24
25     void a()
26     {
27         int x = 25;          // initialized each time a is called
28
29         System.out.println( " \nlocal x in a is " + x +
30                             " after entering a" );
31         ++x;
32         System.out.println( "local x in a is " + x +
33                             " before exiting a" );
34     }
35
36     void b()
37     {
38         System.out.println( " \ninstance variable x is " + x +
39                             " on entering b" );
40         x *= 10;
41         System.out.println( "instance variable x is " + x +
42                             " on exiting b" );
43     }
44 }

```



#### 输出结果:

```

local x in paint is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

instance variable x is 1 on entering b
instance variable x is 10 on exiting b

local x in a is 25 after entering a
local x in a is 26 before exiting a

instance variable x is 10 on entering b
instance variable x is 100 on exiting b

local x in paint is 5

```

图 4.10 一个作用域的例子

程序中将一个实例变量 `x` (第 7 行) 声明并初始化为 1。这个实例变量在任何包含名为 `x` 的变量声明的块中均会隐藏。在 `paint` 方法中, 将一个局部变量 `x` 声明并初始化为 5, 打印该变量以表明在

paint方法中将隐藏实例变量x。程序定义了另外两个方法，每一个都没有参数和返回值。方法a定义了自动变量x并将其初始化为25。当调用a时，打印变量，然后增加其值，并在方法结束前再次打印该变量。在每次调用这个方法时，都将创建自动变量x并初始化为25。方法b没有声明任何变量，因此，当它引用变量x时，使用实例变量x。当调用b时，打印实例变量，然后将其乘以10，在退出方法前再次打印。再一次调用b时，实例变量拥有已修改的值10。最后，程序在paint方法中再次打印局部变量x，显示的结果说明没有方法调用修改x的值，因为这些方法引用了其他作用域中的变量。

## 4.12 递归

我们讨论的程序通常都以严格的、层次化的方式使用一个方法来调用另一个方法。对于某些问题，通过方法来调用自己是很有用的。递归方法（recursive method）是一种直接或通过其他方法间接调用自己的方法，本节和下一节会提供一些简单的递归例子，本书包括了各种递归处理方法。图4.15总结了本书中的递归例子和练习。

我们先从概念上考虑递归，接着考察几个包含递归方法的程序。递归问题的解决方法有许多共性问题。当调用递归方法来解决问题时，该方法实际上只知道如何解决最简单的情况，即基本情况。如果在基本情况下调用方法，那么仅返回一个结果。如果在更复杂的情况下调用方法，那么方法就把问题分成两种概念性部分：一部分方法知道如何处理，而另一部分方法则不知道如何处理。为了使递归成为可能，后一部分必须模拟原始问题，即处理原始问题的较简单或较小的版本。因为这个新问题看起来很像原始问题，这个方法就调用自己的一份新副本，从而在较小的问题上工作。这就是递归调用，也称为递归步骤。递归步骤包括关键字return，因为它的结果将和该方法需要继续处理的部分一起组成一个最终结果，这个结果将传递回原始调用者。

递归步骤在对方法的原始调用仍然继续时处于执行状态，即它还未完成执行。当方法将每个新的子问题分解为两个概念性部分时，递归步骤可能导致另外更多的递归调用。为了使递归最终能够结束，方法每次都使用原始问题的较简单版本来调用自己，一系列越来越小的问题最终会成为基本情况。最后，该方法识别基本情况，向方法的前一个副本返回一个值。经过一系列的返回，直到原始方法向调用者返回最终的值为止。与目前我们所使用的传统问题的解决方式相比，这看上去有些复杂。作为这些概念实际应用的例子，让我们编写一个递归程序来完成一个常见的数学计算。

非负整数 $n$ 的阶乘，写做 $n!$ （读做“ $n$ 的阶乘”），即下列数的乘积：

$$n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$$

1!为1，0!定义为1。例如，5!是乘积 $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ ，等于120。

一个大于等于0的整数number的阶乘可以使用for循环（非递归）进行计算：

```
factorial = 1;
for ( int counter = number; counter >= 1; counter -- )
    factorial*=counter;
```

阶乘方法的递归定义如下所示：

$$n! = n \cdot (n-1)!$$

例如，5!就是 $5 \cdot 4!$ ，即：

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

其计算过程如图 4.11 所示。图 4.11(a) 表明递归调用持续进行到将 1! 计算为 1 时为止，从而终止递归。图 4.11(b) 表明了每次递归调用的结果将返回至调用者，直到计算出最后的结果并将其返回。

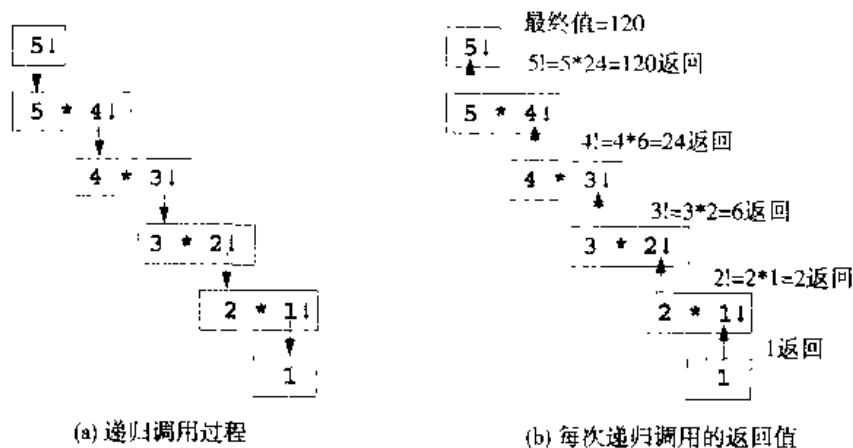


图 4.11 递归计算 5!

图 4.12 的程序使用递归来计算，并打印整数 0 到 10（这里选择数据类型 long，将在稍后解释）的阶乘。递归方法 factorial 首先测试终止条件是否为 true，即 number 小于或等于 1。如果 number 已经小于或等于 1，factorial 就返回 1，无需进一步递归，程序终止。如果 number 仍大于 1，下列语句：

```
return number * factorial( number - 1 );
```

将问题表达为 number 和递归调用 factorial（计算 number - 1 的阶乘）的乘积。注意，factorial(number - 1) 比起原始计算 factorial(number) 稍微简单了一些。

```
1 // Fig. 4.12: FactorialTest.java
2 // Recursive factorial method
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class FactorialTest extends Applet {
7     public void paint( Graphics g )
8     {
9         int yPosition = 25;
10
11         for ( long i = 0; i <= 10; i++ ) {
12             g.drawString( i + "! = " + factorial( i ),
13                 25, yPosition );
14             yPosition += 15;
15         }
16     }
17
18     // Recursive definition of method factorial
19     public long factorial( long number )
20     {
21         if ( number <= 1 ) // base case
22             return 1;
```

```
23         else
24             return number * factorial( number - 1 );
25     }
26 }
```



图 4.12 使用一个递归方法来计算阶乘

`factorial` 方法已经声明接收一个 `long` 类型的参数，并返回一个 `long` 类型的结果。正如我们在图 4.12 中看到的，阶乘值会迅速增大。我们选择了数据类型 `long`，于是程序可以计算大于  $20!$  的阶乘。不过，`factorial` 方法产生大额数值的速度非常快，有时即使是 `long` 类型也无法在变量溢出之前打印出许多阶乘的值。

正如我们在练习中将要探讨的那样，`float` 和 `double` 可能常用来计算较大数值的阶乘，这体现了大多数编程语言的弱点，即语言不易扩展以处理各种应用的特定需求。我们将在本书后面的章节中看到，Java 是一种可扩展语言，允许我们创建需要的任意大的整数。

#### 常见编程错误 4.13

当需要结果而忘记从一个递归方法返回一个值时会产生语法错误。

#### 常见编程错误 4.14

忽略了基本情况，或者不正确地编写了递归步骤而未包含基本情况，将导致无限递归，最终耗尽内存，这类似于一个无限循环（非递归）的问题。可能也会由于提供了一个不期望的输入而引起无限递归。

## 4.13 使用递归的例子：斐波纳契数列

斐波纳契（Fibonacci）数列：

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

以 0 和 1 开始，其特性为每个后续的斐波纳契数皆为前两个斐波纳契数之和。

这种数列在自然界中很常见，特别是描述了一种螺旋形状。相邻斐波纳契数之比收敛于常量值 1.618...。这个数字在自然界中反复出现并称其为黄金比例或黄金分割。人们一般认为黄金分割会产生最佳的欣赏效果，建筑师们常常按照黄金分割来设计窗户、房间和建筑物的长宽比例，明信片也常常按照黄金分割进行设计。

斐波纳契数列可以递归定义为：

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

图4.13使用方法 fibonacci 递归计算了第  $n$  个斐波纳契数。注意，斐波纳契数增长得很快，因此我们在 fibonacci 方法中将参数类型和返回类型均定义为数据类型 unsigned long。在图4.13中，每对输出行显示运行了一次程序的结果。

```
1 // Fig. 4.13: FibonacciTest.java
2 // Recursive fibonacci method
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class FibonacciTest extends Applet {
7     Label numLabel, resultLabel;
8     TextField num, result;
9
10    public void init()
11    {
12        numLabel = new Label( "Enter an integer and press return" );
13        num = new TextField( 10 );
14        resultLabel = new Label( "Fibonacci value is" );
15        result = new TextField( 15 );
16        result.setEditable( false );
17
18        add( numLabel );
19        add( num );
20        add( resultLabel );
21        add( result );
22    }
23
24    public boolean action( Event e, Object o )
25    {
26        long number, fibonacciVal;
27
28        number = Long.parseLong( num.getText() );
29        showStatus( "Calculating ..." );
30        fibonacciVal = fibonacci( number );
31        showStatus( "Done." );
32        result.setText( Long.toString( fibonacciVal ) );
33        return true;
34    }
35
36    // Recursive definition of method fibonacci
37    long fibonacci( long n )
38    {
39        if ( n == 0 || n == 1 ) // base case
40            return n;
41        else
42            return fibonacci( n - 1 ) + fibonacci( n - 2 );
43    }
44 }
```



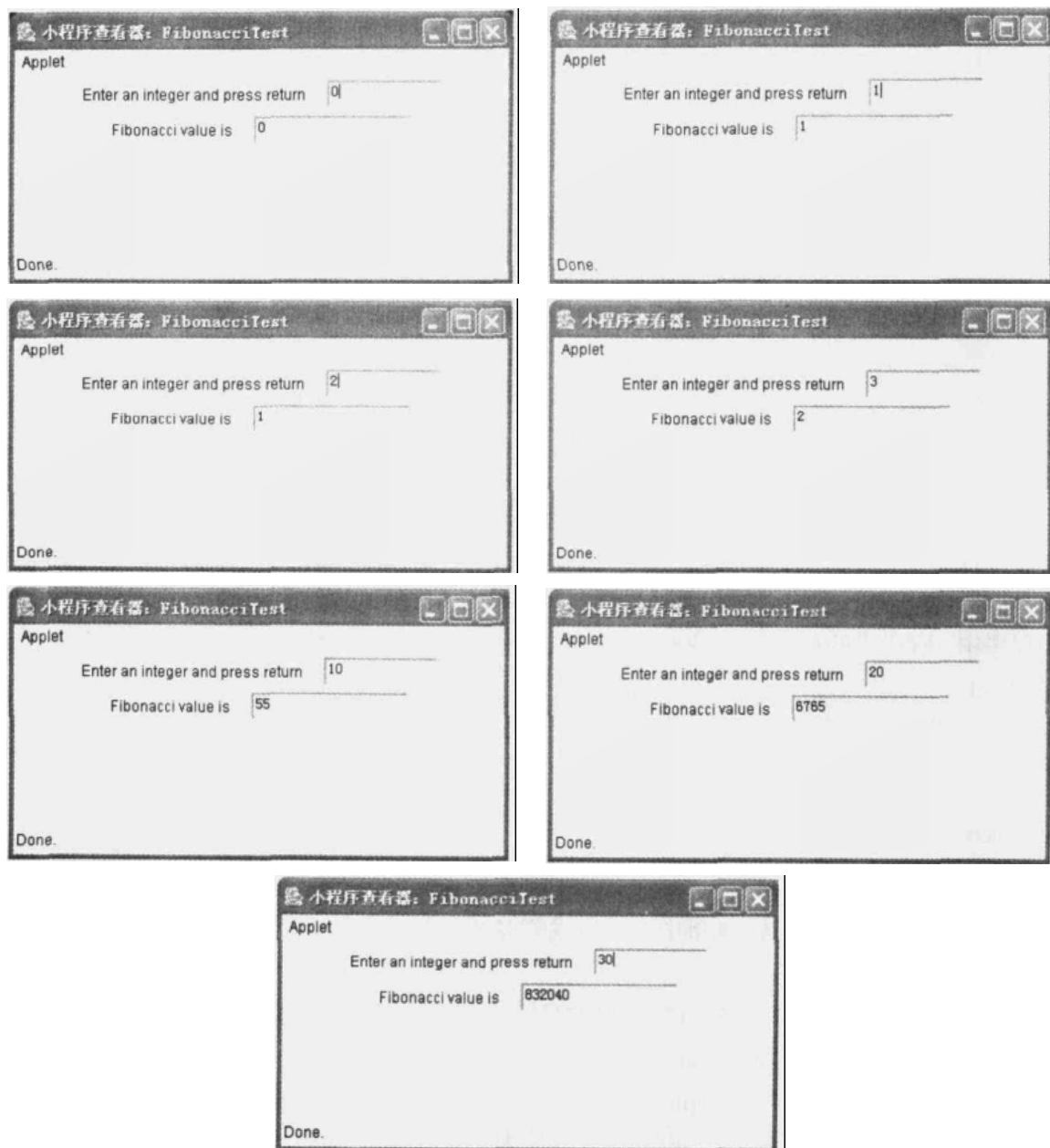


图 4.13 递归产生斐波纳契数

从 action 调用 fibonacci 不是一个递归调用，但后面对 fibonacci 的所有调用都是递归的。每次激活 fibonacci 方法，它将立即测试基本情况—— $n$  等于 0 或 1。如果结果为真，则返回  $n$ 。有趣的是，如果  $n$  大于 1，那么递归步骤将产生两个递归调用，每个都将面临比对 fibonacci 的原始调用更简单的问题。图 4.14 表明了 fibonacci 方法是如何计算 fibonacci(3) 的，我们将 fibonacci 缩写为  $f$ ，以使该图的可读性更强。

在图 4.14 中，我们可以发现一些有趣的问题，即 Java 编译器按照什么顺序来计算运算符两边的操作数。这是由于该顺序与运算符的顺序不同而产生的问题，称为由运算符优先级规则支配的顺序。从图 4.14 可以看出，在计算  $f(3)$  时，将会有两个递归调用，但这些调用的顺序是如何决定的呢？大多数程序员简单地假定从左到右计算操作数，在 Java 中也确实如此。

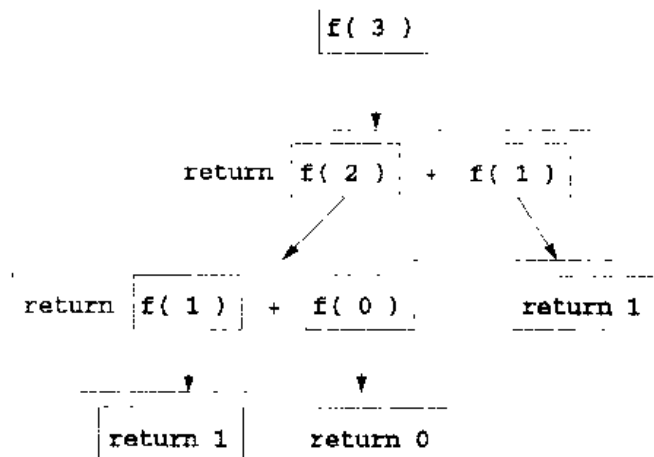


图 4.14 对 fibonacci 方法的递归调用

奇怪的是，C 和 C++ 语言（许多 Java 的特征来自于它们）并未指明大多数运算符（包括 +）的操作数计算顺序。因此，程序员不能假设这些语言中这些调用执行的顺序。事实上调用可能先执行  $f(2)$ ，然后才执行  $f(1)$ ；或者按相反的顺序执行，先执行  $f(1)$ ，然后才是  $f(2)$ 。在本例程序和大多数其他程序中，所产生的最终结果是一样的。但在某些程序中，一个操作数的计算可能会产生副作用（side effect），从而影响到表达式的结果。

Java 语言中运算符的操作数计算顺序是从左到右的。因此，事实上先调用方法  $f(2)$ ，然后调用方法  $f(1)$ 。

#### 编程技巧 4.8

不要编写依赖于运算符的操作数计算顺序的表达式，这常常会导致程序难以阅读、调试、修改和维护。

另一个要注意的问题是，本例中用来生成斐波纳契数列的递归程序的顺序。每次对没有匹配到基本情况之一（即 0 或 1）的 fibonacci 方法的调用，都会导致两个以上对 fibonacci 方法的递归调用，这种情况很快就会失控。使用图 4.13 中的程序计算 fibonacci 值 20，需要调用 fibonacci 方法 21 891 次，计算 fibonacci 值 30，需要调用 fibonacci 方法 2 692 537 次。

当试图使用更大的值时，applet 要计算的、每个相邻的斐波纳契数都会导致计算时间和调用 fibonacci 方法次数的实质性增长。例如，计算值为 31 的斐波纳契数需要 4 356 617 次调用，而值为 32 的斐波纳契数需要 7 049 155 次调用。可以看出，对 fibonacci 的调用次数在迅速增长——在 30 和 31 之间的 fibonacci 调用增加了 1 664 080 次，在 31 和 32 之间的 fibonacci 调用增加了 2 692 538 次。31 和 32 之间的 fibonacci 的调用次数为 30 和 31 之间的 fibonacci 调用的 1.5 倍。这一性能上的问题足以使世界上功能最强大的计算机望而却步！在复杂性理论的领域中，计算机科学家正在研究算法相对于要完成的工作量的实现难度。

#### 性能提示 4.2

避免编写斐波纳契式的递归程序，这会导致调用的指数型“爆炸”。

## 4.14 递归与迭代

在前面几节中，我们研究了两个可通过递归或迭代来实现的方法。在本节中我们将比较两种方法，并讨论为什么在特定场合下程序员选择其中一种而非另一种。

递归和迭代都是基于单一的控制结构：迭代使用一种重复结构（如 for、while 或 do/while），递归则使用一种选择结构（如 if、if/else 或 switch）。递归和迭代都涉及重复：迭代显式地使用一个重复结构，递归则进行重复的方法调用。迭代和递归各有一个终止测试：迭代是当迭代条件失效时终止；递归则是在识别基本情况后终止。迭代利用计数器来控制重复次数，而递归则是逐步接近终止情况：迭代跟踪修改一个计数器，直到计数器达到使循环条件失效的值；递归则跟踪产生原始问题的简化版本，直到达到基本情况。迭代和递归都可以无限进行：无限迭代发生在迭代的循环条件测试永远不为 false 的情况下；无限递归则发生在递归步骤最终不能收敛于基本情况，从而无法缩小求解目标。

递归有许多缺点，它不断激活方法调用的机制，从而增加方法调用的负担，这对处理器时间和存储空间而言都是代价高昂的。每次递归调用都将创建方法的另一个副本（实际上仅是方法的变量），这可能占用相当大的内存。迭代一般发生在一个方法内，于是重复的方法调用和过度的内存分配都可以忽略。那么为什么还要选择递归呢？

#### 软件工程视点 4.10

任何可用递归解决的问题也能使用迭代（非递归）解决。当递归方法可以更自然地反映问题，并且产生易于理解和调试的程序时，就可以选择递归方法而非迭代方法。另一个选择递归方案的原因可能是迭代方案不够清楚。

#### 性能提示 4.3

在要求高性能的情况下避免使用递归。递归调用既花时间又消耗额外的内存。

#### 常见编程错误 4.15

使用一种非递归方法直接调用自己或通过另一种方法间接调用自己。

大多数关于编程的教材中，递归概念的出现都要晚于本书。我们觉得递归是一个十分丰富和复杂的课题，因此最好提早介绍并且在本书的其余部分扩展这个问题。图 4.15 总结了本书中关于递归的例子和练习。

章名	关于递归的例子和练习
第4章	阶乘方法
	斐波纳契方法
	最大公约数
	两个整数的和
	两个整数相乘
	一个整数的整数次幂
	汉诺塔
	可视化递归
	一个数组元素的和
	打印一个数组
第5章	逆向打印一个数组
	检查一个字符串是否为回文
	一个数组中的最小值
	选择排序
	八皇后
	线性查找
	二分查找
	快速排序
	走迷宫

(续表)

章名	关于递归的例子和练习
第 8 章	逆向打印通过键盘输入的字符串
第 17 章	链表插入
	链表删除
	查找一个链表
	逆向打印链表二叉树的插入
	二叉树的前序遍历
	二叉树的中序遍历
	二叉树的后序遍历

图 4.15 本书中所有关于递归的例子和练习

让我们考虑在本书中不断重复的一些观点：良好的软件工程是重要的，较高的性能也是重要的，但是这些目标常常彼此矛盾。良好的软件工程是使开发较大和复杂的软件变得更易于管理的关键，而这些系统中的高性能是实现未来更高级系统的关键，同时高性能对硬件提出越来越高的计算要求。应该尽量满足哪一方呢？

**软件工程视点 4.11**

将程序模块化会以一种清晰、层次化的方式推进良好的软件工程，但它是有代价的。

**性能提示 4.4**

与一个“长篇大论”的（即连成一片）、没有方法的程序相比，一个高度模块化的程序拥有大量的方法调用，并且这些调用占用执行时间和计算机的内存空间。但“长篇大论”的程序难以编程、测试、调试、维护和改进。

因此要有条理地将程序模块化，并牢记在高性能和良好的软件工程之间要保持平衡。

## 4.15 方法重载

Java 可以定义几个同名方法，只要这些方法有不同的参数集合（基于参数个数、参数类型和参数的顺序），这就是所谓的方法重载（method overloading）。当调用重载的方法时，Java 编译器通过考察参数个数、类型和顺序来选择合适的方法，方法重载通常用于创建完成任务相似、但有不同数据类型的一个同名方法。

**编程技巧 4.9**

重载方法可以完成密切相关的任务，从而使程序易读且易于理解。

图 4.16 中使用重载的方法 square 来计算 int 类型和 double 类型的数值的平方。

重载的方法通过它们的特征（signature）——方法的名字和其参数类型的组合来进行区分。如果 Java 的编译器在编译时只依据方法名字，那么图 4.16 中的代码将会产生混淆，编译器将不知道如何区分两个 square 方法。从逻辑上分析，编译器将使用较长的“组合的”或“修饰过的”名字，它包括原方法名、每个参数类型和参数的准确顺序，以确定一个类中的方法在这个类中是否是唯一的。

```
1 // Fig. 4.16: MethodOverload.java
2 // Using overloaded methods
```

```
3   import java.awt.Graphics;
4   import java.applet.Applet;
5
6   public class MethodOverload extends Applet {
7       public void paint( Graphics g )
8       {
9           g.drawString( "The square of integer 7 is " + square( 7 ),
10                        25, 25 );
11           g.drawString( "The square of double 7.5 is " +
12                        square( 7.5 ), 25, 40 );
13       }
14
15       int square( int x )
16       {
17           return x * x;
18       }
19
20       double square( double y )
21       {
22           return y * y;
23       }
24   }
```

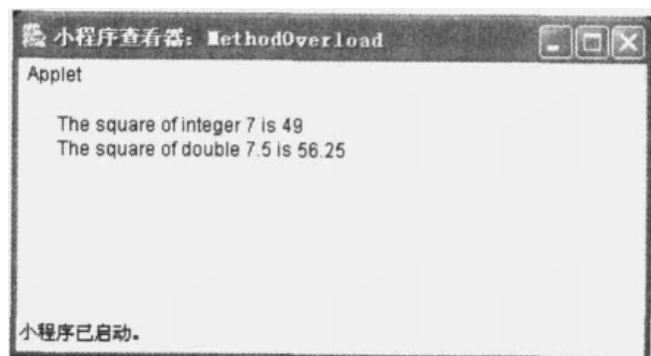


图 4.16 使用重载的方法

以图 4.16 为例，编译器也许使用逻辑名“square of int”指明定义 int 参数的 square 方法，而使用“square of double”指明定义 double 参数的 square 方法。如果一个方法 foo 的定义如下所示：

```
void foo( int a , float b )
```

则编译器可能会使用逻辑名“foo of int and float”。如果参数定义为：

```
void foo( float a , int b )
```

则编译器可能会使用逻辑名“foo of float and int”。注意，参数的顺序对编译器而言是很重要的，前面两个 foo 方法对编译器而言是截然不同的。

到目前为止，由编译器使用的逻辑名没有提及方法的返回类型。这是因为方法不能由返回类型进行区分。图 4.17 的程序给出了由于两个方法有相同的特征值和不同的返回类型而产生的编译错误。重载的方法可以有不同的返回类型，但必须要有不同的参数表。另外，重载方法也不必有相同数目的参数。

**常见编程错误 4.16**

使用相同的参数表 and 不同的返回值创建重载方法会产生语法错误

```

1 // Fig. 4.17: MethodOverload.java
2 // Overloaded methods with identical signatures and
3 // different return types.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class MethodOverload extends Applet {
8     int square( double x )
9     {
10         return x * x;
11     }
12
13     double square( double y )
14     {
15         return y * y;
16     }
17 }

```

**输出结果:**

```

MethodOverload.java:13:Methods can't be redefined with a
    different return type:double square(double) was
    int square(double)
    double square(double y)
        ^
1 error

```

图 4.17 使用相同参数表 and 不同返回类型的  
重载方法所产生的编译器错误消息

## 4.16 Applet 类的方法

从开始学习本书到现在,我们已经编写了许多 applet 程序,但还没有讨论 Applet 类中的关键方法,这些关键方法在 applet 执行时将自动调用。图 4.18 列出了 Applet 类中的关键方法,并说明了何时调用这些方法以及每个方法的实现目的。

除非在 applet 的类定义中提供了一个定义,否则不会调用这些由 Java API 定义的 Applet 方法。如果要在定义的 applet 中使用这些方法,就必须按照图 4.18 所示的方式来定义方法的首行。否则,在 applet 的执行期间将不会自动调用该方法。

**常见编程错误 4.17**

为 Applet 方法 init、start、paint、stop 或 destroy 之一提供定义,但是不匹配图 4.18 所示的方法首部,这样会导致在 applet 的执行期间不能自动调用该方法。

方法	方法调用时机及其目的
public void init()	当加载一个 applet 并执行时,这个方法仅由 appletviewer 或浏览器调用一次。它完成 applet 的初始化。此时完成的典型动作包括:初始化实例变量和 Applet 类的 GUI 构件,加载声音或图像(详细内容请参见第 14 章),并创建线程(详细内容请参见第 13 章)

(续表)

方法	方法调用时机及其目的
<code>public void start()</code>	这个方法在 <code>init</code> 方法执行完后, 当每次浏览器的用户返回到 applet 所在的 HTML 页面上时 (浏览另一个 HTML 页面之后) 调用。这个方法要完成任何 applet 第一次加载到 appletviewer 或浏览器时必须完成的任务, 还必须完成每次重新访问 applet 所在的 HTML 页面时执行的任务。此时必须完成的典型动作包括启动一个动画 (请参见第 14 章) 和启动其他执行线程 (请参见第 13 章)
<code>public void paint( Graphics g )</code>	这个方法在执行完 <code>init</code> 方法并且启动方法已开始在 applet 上绘图之后调用; 同时, 在每次 applet 需要重新绘图时也将自动调用该方法。例如, 如果 appletviewer 或浏览器的用户在他的计算机上用另一个打开的窗口覆盖了这个 applet, 然后又要显示 applet, 则重新调用 <code>paint</code> 方法。此时完成的典型动作是利用自动传入 <code>paint</code> 方法的 <code>Graphics</code> 对象 <code>g</code> 进行绘图
<code>public void stop()</code>	这个方法在 applet 停止执行后, 一般在浏览器的用户离开 applet 所在的 HTML 页面时调用。这个方法完成任何用来挂起 applet 执行的任务。典型的动作是停止动画播放和线程的执行
<code>public void destroy()</code>	这个方法在 applet 从内存中删去, 一般是在浏览器的用户退出浏览时调用, 主要完成任何用来取消分配给该 applet 的资源的任务。此时完成的典型动作是终止线程 (详细内容请参见第 13 章)

图 4.18 在 applet 执行时自动调用的 Applet 方法

许多 applet 程序员也对 `resize` 和 `repaint` 方法产生了兴趣。当通过一个 HTML 页面激活 applet 程序时, applet 宽和高的像素值在 HTML 代码 (如图 1.3 所示) 中就已经定义了。在执行期间可以通过激活 `resize` 方法来改变 applet 的大小。例如, 下列语句:

```
resize ( 200, 400 );
```

使 applet 的宽为 200 像素而高为 400 像素。这条语句一般放在 applet 的 `init` 方法中。

applet 的 `paint` 方法一般为自动调用。如果读者想改变 applet 的外观并以此掌握用户与 applet 的交互, 那么应该怎么做呢? 在这种情况下, 可以直接调用 `paint` 方法。但是要调用 `paint` 方法, 就必须向其传递该方法需要的 `Graphics` 参数, 这样就会产生问题。前面没有将 `Graphics` 对象传递给 `paint` 方法 (将在第 14 章讨论这一点), 因此 Applet 类又提供了 `repaint` 方法。下列语句:

```
repaint ( );
```

激活了另一个名为 `update` 的方法并传递了 `Graphics` 对象。

方法 `update` 擦除过去在 applet 中所绘制的所有图形, 然后激活 `paint` 方法并传递 `Graphics` 对象。`repaint` 和 `update` 方法将在第 14 章详细讨论。

## 小结

- 开发和维护大型程序的最佳方法是将其分为几个较小的程序模块, 每个模块都比原始程序更易于管理。Java 中的模块分为类和方法。
- 一个方法由该方法的调用激活。方法调用要使用方法名, 并且提供被调用方法用来完成任务的信息 (实参)。
- 在程序中通过写下其名字并后接括号中的参数表来激活方法。
- 方法的实参可能是常量、变量或表达式。
- 一个局部变量仅在方法定义中可见, 方法不能知道其他方法 (包括局部变量) 的实现细节。
- 方法定义的一般格式为:

返回值类型 方法名 ( 参数表 )

```
{  
    声明和语句  
}
```

返回值类型说明了返回给调用方法的值的类型。如果一个方法不返回一个值,则返回值类型为空 (void)。方法名是任何合法的标识符,形参表是一个由逗号分隔,包括将传至方法中的变量声明的列表。如果一个方法不接收任何值,则形参表为空。方法体为一组构成方法的声明和语句。

- 传入方法的实参应当同方法定义中的形参在数目、类型和顺序上匹配
- 当程序遇到方法时,将控制从调用点转移到调用的方法,并且执行该方法,然后将控制返回给调用者。
- 一个被调用方法可以使用三种方式将控制返回给调用者:如果方法没有返回一个值,则在达到终止方法的右花括号时返回控制,或者执行下列语句:

```
return;
```

如果方法的确要返回一个值,则执行下列语句:

```
return expression;
```

这将返回表达式的值。

- Math.random 方法产生一个从 0.0 到 1.0 (不包括 1.0) 的双精度值
- 由 Math.random 产生的值可以对其进行比例缩放和移位,从而产生特定范围内的值。
- 通用的比例缩放和移位一个随机数的等式为:

```
n = a + (int) ( Math.random( ) * b );
```

其中 a 是位移值 (它等于需要的连续整数范围内的开始值),而 b 是比例因子 (它等于需要的连续整数范围的宽度)。

- 每个变量标识符都具有生存期和作用域。一个标识符的生存期决定该标识何时存在于内存中,一个标识符的作用域是指在程序中可以引用该标识符的范围。
- 在方法中表示局部变量的标识符 (即方法体中声明的形参和变量)具有自动生存期。自动生存期变量在进入声明它们的块时创建,当块激活时存在,在退出块时删除。
- Java 也有静态生存期标识符,静态生存期变量和方法将一直存在,从它们所在的类加载到内存开始,直到程序终止。对于静态生存期变量,在它们的类加载到内存时一次性分配和初始化存储空间。对于静态生存期方法,在将它们的类加载到内存时方法的名字一直存在。
- 标识符的作用域分为类作用域和块作用域。
- 在方法之外声明的变量具有类作用域,这类标识符在类的所有方法中都是“已知”的。
- 在块中声明的标识符具有块作用域,块作用域在块的终止右花括号 (}) 处结束。
- 在方法开始处声明的局部变量具有同方法形参一样的作用域,方法将其视为局部变量。
- 递归方法就是直接或间接调用自己的方法。
- 如果使用基本情况调用递归方法,则方法简单地返回一个值。如果使用一个更复杂的问题调用方法,则该方法就将其分解为两个或更多的概念性部分:一部分为方法知道如何处理的问题,而另一部分是原始问题的简化版本。因为这个新问题看起来类似于原始问题,所以方法将在较小的问题上继续进行递归调用。



- 为了使递归终止,每次递归方法都使用原始问题的简化版本来调用自己,在一系列越来越小的问题中必须包括基本情况。当该方法识别基本情况时,其结果就返回到前一个方法调用中,一系列的返回保证了朝目标前进,直到方法的原始调用返回其最后结果。
- 迭代和递归都是单一的控制结构:迭代使用一个重复结构,递归使用了一个选择结构。
- 迭代和递归都包括重复过程:迭代显式地使用了一个重复结构,递归通过反复调用来实现重复过程。
- 迭代和递归各包含一个终止测试:迭代在循环条件失效时终止,递归在识别基本情况时终止。
- 递归不断激活调用机制,因而增加了方法调用的负担,这对于处理器时间和内存空间而言,其代价都是昂贵的。
- 一个没有返回值的方法声明有一个 void 返回类型。如果试图从不返回值的方法中返回一个值或在调用该方法时使用方法的结果,则编译器会报告错误。
- final 修饰符创建了“常量变量”。一个常量必须在变量声明时初始化,而且在以后不能修改。常量变量也称为命名常量或只读变量。
- 可以定义同名而具有不同形参表(基于形参类型、形参数目和形参顺序)的方法,这就是方法重载。当调用一个重载的方法时,编译器通过检查调用的实参来选择合适的方法。
- Applet 类的 paint 方法在 init 方法完成之后以及 start 方法开始绘制 applet 时调用。paint 方法也在每次 Applet 类需要重新绘图时自动调用。
- Applet 类的 stop 方法在 applet 应当挂起时执行;一般情况下,也就是指浏览器的用户退出浏览时调用。

## 术语

argument in a method call 方法调用中的实参

automatic duration 自动生存期

automatic variable 自动变量

base case in recursion 递归中的基本情况

block 块

block scope 块作用域

call a method 调用方法

called method 被调用的方法

caller 调用者

calling method 调用的方法

class 类

class scope 类作用域

coercion of arguments 形参的强制类型转换

constant variable 常量变量

copy of a value 值的副本

destory method destroy 方法

divide and conquer 分而治之,各个击破

duration 生存期

element of chance 机会元素

factorial method 阶乘方法

file scope 文件作用域

final

init method init 方法

invoke a method 激活一个方法

iteration 迭代

Java API (Java class library) Java API (Java 类库)

local variable 局部变量

Math class methods Math 类的方法

Math.E

Math.PI

Math.random method Math.random 方法

method 方法

method call 方法调用

method declaration	方法声明	reference types	引用类型
method definition	方法定义	repaint method	repaint 方法
method overloading	方法重载	resize method	resize 方法
mixed-type expression	混合类型表达式	return	
modular program	模块化程序	return-value-type	返回值类型
named constant	命名常量	scaling	比例缩放
overloading	重载	scope	作用域
paint method	paint 方法	shifting	移位
parameter in a method definition	方法定义中的形参	side effects	副作用
programmer-defined method	程序员定义的方法	signature	特征
promotion rules	提升规则	simulation	模拟
random number generation	随机数产生	software engineering	软件工程
read-only variable	只读变量	software resuability	软件可重用性
recursion	递归	start method	start 方法
recursion step	递归步骤	static storage duration	静态存储生存期
recursion call	递归调用	stop method	stop 方法
recursive method	递归方法	update method	update 方法
reference parameter	引用参数	void	

## 自测练习

### 4.1 填空:

- a) Java 中的程序模块称为 \_\_\_\_\_ 和 \_\_\_\_\_。
- b) 方法利用 \_\_\_\_\_ 激活。
- c) 一个只在定义它的方法中可见的变量称为 \_\_\_\_\_。
- d) 在被调用方法中, \_\_\_\_\_ 语句可用来将一个表达式的值传递回调用方法。
- e) 关键字 \_\_\_\_\_ 用于在方法首部中指明一个方法没有返回值。
- f) 标识符的 \_\_\_\_\_ 是该标识符可在程序中使用的部分。
- g) 从被调用方法向调用者返回控制的三种形式分别为 \_\_\_\_\_。
- h) \_\_\_\_\_ 方法仅在 applet 开始执行时激活一次。
- i) \_\_\_\_\_ 方法用于产生随机数。
- j) \_\_\_\_\_ 方法在一个浏览器的用户每次重新访问 applet 所在的 HTML 页面时激活。
- k) 激活 \_\_\_\_\_ 方法用于在 applet 中绘图。
- l) 在一个块中或在一个方法的形参表中声明的变量都假定有 \_\_\_\_\_ 生存期。
- m) 激活 \_\_\_\_\_ 方法用于改变 applet 执行中的宽和高。
- n) \_\_\_\_\_ 方法激活 applet 的 update 方法, 它又激活 applet 的 paint 方法。
- o) 方法在 \_\_\_\_\_ 每次一个浏览器的用户离开 applet 所在的 HTML 页面时激活。
- p) 直接或间接调用自己的方法是一个 \_\_\_\_\_ 方法。
- q) 递归方法有两个概念性部分: - 一个提供了通过测试 \_\_\_\_\_ 情况来终止递归的手段, 另一个将问题表示为比原始调用更简单的递归调用。
- r) 在 Java 中, 不同的方法可以同名并且可以有不同的实参类型和 / 或实参数目。这就是

方法 \_\_\_\_\_

s) \_\_\_\_\_ 修饰符用于声明只读变量

4.2 根据下面的程序, 指出下面每个元素的作用域(类作用域或块作用域)。

a) 变量 `x`。

b) 变量 `y`。

c) 方法 `cube`。

d) 方法 `paint`。

e) 变量 `yPos`。

```
public class CubeTest extends Applet {
    int x;

    public void paint( Graphics g )
    {
        int yPos = 25;

        for ( x = 1; x <= 10; x++ ) {
            g.drawString( cube( x ) , 25, yPos );
            yPos += 5;
        }
    }

    public int cube ( int y ) {
        return y * y * y;
    }
}
```

4.3 编写一个应用程序, 测试图 4.2 所示的 `Math` 类方法调用的例子是否产生了预期结果。

4.4 给出下面方法的方法首部:

a) `hypotenuse` 方法接收两个双精度浮点数参数 `side1` 和 `side2`, 返回一个双精度浮点数结果。

b) `smallest` 方法接收三个整数 `x`、`y`、`z`, 并返回一个整数。

c) `instructions` 方法不接收任何参数也没有返回值(提示: 这类指令通常用于向用户显示指令)。

d) `intToFloat` 方法接收一个整型参数 `number` 并返回一个浮点数结果。

4.5 在下面每个程序段中找出错误并解释如何才能改正错误:

```
a) int g() {
    System.out.println( "Inside method g" );
    int h() {
        System.out.println( "Inside method h" );
    }
}

b) int sum( int x, int y ) {
    int result;

    result = x + y;
}
```

```

c) int sum( int n ) {
    if ( n==0 )      return 0;
    else
        n = sum( n-1 ) ;
}

d) void f( float a ) {
    float a;

    System.out.println( a ) ;
}

e) void product( ) {
    int a = 5, b = 5, c = 4, result;
    result = a * b * c;
    System.out.println( "Result is " + result ) ;
    return result;
}

```

- 4.6 编写一个完整的 Java 程序，提示用户输入球体的 double 半径，调用 sphereVolume 方法，使用下面的公式计算并显示球体的体积：

$$\text{volume} = (4/3) * \text{Math.PI} * \text{Math.pow}(\text{radius}, 3)$$

用户应当通过一个文本字段输入半径。

## 自测练习答案

- 4.1 a) 方法，类。b) 方法调用。c) 局部变量。d) return。e) void。f) 作用域。g) “return,”，“return 表达式;”，或是遇到方法的右花括号。h) init。i) Math.random。j) start。k) paint。l) 自动。m) resize。n) repaint。o) stop。p) 递归。q) 基本。r) 重载。s) final。
- 4.2 a) 类作用域。b) 块作用域。c) 类作用域。d) 类作用域。e) 块作用域。
- 4.3 下面的解决方案展示了图 4.2 中的 Math 类方法。

```

1 //Exercise 4.3 : MathTest.java
2 //Testing the Math class methods
3
4 public class MathTest {
5     public static void main( String args[] )
6     {
7         System.out.println( "Math.abs( 23.7 ) = " + Math.abs( 23.7 ) );
8
9         System.out.println( "Math.abs( 0.0 ) = " + Math.abs( 0.0 ) );
10
11        System.out.println( "Math.abs( -23.7 ) = " + Math.abs( -23.7 ) );
12
13        System.out.println( "Math.ceil( 9.2 ) = " + Math.ceil( 9.2 ) );
14
15        System.out.println( "Math.ceil( -9.8 ) = " + Math.ceil( -9.8 ) );
16
17        System.out.println( "Math.cos( 0.0 ) = " + Math.cos( 0.0 ) );

```

```
18
19     System.out.println( "Math.exp( 1.0 ) = " + Math.exp( 1.0 ) );
20
21     System.out.println( "Math.exp( 2.0 ) = " + Math.exp( 2.0 ) );
22
23     System.out.println( "Math.floor( 9.2 ) = " + Math.floor( 9.2 ) );
24
25     System.out.println( "Math.floor( -9.8 ) = " + Math.floor( -9.8 ) );
26
27     System.out.println( "Math.log( 2.71828 ) = " + Math.log( 2.71828 ) );
28
29     System.out.println( "Math.log( 7.389056 ) = " + Math.log( 7.389056 ) );
30
31     System.out.println( "Math.max( 2.3, 12.7 ) = " + Math.max( 2.3, 12.7 ) );
32
33     System.out.println( "Math.max( -2.3, -12.7 ) = " + Math.max( -2.3, -12.7 ) );
34
35     System.out.println( "Math.min( 2.3, 12.7 ) = " + Math.min( 2.3, 12.7 ) );
36
37     System.out.println( "Math.min( -2.3, -12.7 ) = " + Math.min( -2.3, -12.7 ) );
38
39     System.out.println( "Math.pow( 2, 7 ) = " + Math.pow( 2, 7 ) );
40
41     System.out.println( "Math.pow( 9, .5 ) = " + Math.pow( 9, .5 ) );
42
43     System.out.println( "Math.sin( 0.0 ) = " + Math.sin( 0.0 ) );
44
45     System.out.println( "Math.sqrt( 25.0 ) = " + Math.sqrt( 25.0 ) );
46
47     System.out.println( "Math.tan( 0.0 ) = " + Math.tan( 0.0 ) );
48
49 }
50 }
```

**输出结果:**

```
Math.abs( 23.7 ) = 23.7
Math.abs( 0.0 ) = 0
Math.abs( -23.7 ) = 23.7
Math.ceil( 9.2 ) = 10
Math.ceil( -9.8 ) = -9
Math.cos( 0.0 ) = 1
Math.exp( 1.0 ) = 2.71828
Math.exp( 2.0 ) = 7.389056
Math.floor( 9.2 ) = 9
Math.floor( -9.8 ) = -10
Math.log( 2.718282 ) = 1
Math.log( 7.389056 ) = 2
Math.max( 2.3, 12.7 ) = 12.7
Math.max( -2.3, -12.7 ) = -2.3
Math.min( 2.3, 12.7 ) = 2.3
Math.min( -2.3, -12.7 ) = -12.7
Math.pow( 2, 7 ) = 128
Math.pow( 9, .5 ) = 3
Math.sin( 0.0 ) = 0
Math.sqrt( 25.0 ) = 5
Math.tan( 0.0 ) = 0
```

- 4.4 a) double hypotenuse( double side1, double side2 )  
 b) int smallest( int x, int y, int z )  
 c) void instructions( )  
 d) float intToFloat( int number )

- 4.5 a) 错误：方法 h 在方法 g 中定义。

改正：从 g 的定义中去掉 h 的定义

- b) 错误：认为方法返回一个整数，但实际上没有

改正：删除变量 result 并将下面的语句放在方法中：

```
return x + y;
```

- c) 错误：n + sum( n - 1 ) 的结果未返回，sum 返回一个不正确的结果。

改正：在 else 子句中将其重写为下面的语句：

```
return n + sum( n-1 ) ;
```

- d) 错误：参数表的右括号后有分号，并且在方法中重新定义了参数 a。

改正：删除参数表右括号后的分号，删除声明“float a;”。

- e) 错误：当方法声明没有返回值时却返回了一个值

改正：删除 return 语句。

- 4.6 下面的方案以用户输入的半径来计算球体的体积

```

1  //Ex. 4.6 : SphereTest.java
2  import java.applet.Applet;
3  import java.awt.*;
4
5  public class SphereTest extends Applet {
6      Label prompt;
7      TextField input;
8
9      public void init( )
10     {
11         prompt = new Label( "Enter sphere radius:" ) ;
12         input = new TextField( 10 ) ;
13         add( prompt ) ;
14         add( input ) ;
15     }
16
17     public boolean action( Event e, Object o )
18     {
19         Double val = new Double( o.toString( ) );
20         double radius = val.doubleValue( ) ;
21         showStatus( "Volume is " + Double.toString( sphereVolume( radius ) ) );
22
23         return true;
24     }
25
26     public double sphereVolume( double radius )
27     {
28         double volume;
29         volume = ( 4/3 ) * Math.PI * Math.pow( radius, 3 ) ;
30         return volume;
31     }
32 }
```

## 练习

4.7 给出下面每条语句执行后  $x$  的值。

- a)  $x = \text{Math.abs}(7.5)$  ;
- b)  $x = \text{Math.floor}(7.5)$  ;
- c)  $x = \text{Math.abs}(0.0)$  ;
- d)  $x = \text{Math.ceil}(0.0)$  ;
- e)  $x = \text{Math.abs}(-6.4)$  ;
- f)  $x = \text{Math.ceil}(-6.4)$  ;
- g)  $x = \text{Math.ceil}(-\text{Math.abs}(-8 + \text{Math.floor}(-5.5)))$  ;

4.8 一个停车场对3小时以内的停车最少收费是\$2.00。如果超过3小时则按每小时另收\$0.50来计费，不足1小时按1小时算。24小时停车的收费是\$10.00。假定没有汽车一次停车超过24小时，编写一个 applet，为每一个在该停车场停车的用户计算并显示停车费用。程序中需要为每个用户输入小时数，运行结果中应当显示目前客户的费用。并且计算和显示前一天收入的总和。程序中应当使用 `calculateCharges` 方法来决定每个客户的费用。使用自测练习 4.6 中介绍的方法从一个文本字段中读取双精度值。

4.9 `Math.floor` 方法的作用是将一个值取整为最接近的整数。例如，下列语句：

```
y = Math.floor( x + .5 ) ;
```

将  $x$  取整为最接近的整数，并将结果赋给  $y$ 。编写一个 applet，读入双精度值并使用上面的语句将每个这样的数取整为最近的整数。对于处理过的每个整数，同时显示其原始值和取整后的数。使用自测练习 4.6 中介绍的方法从一个文本字段中读取双精度值。

4.10 `Math.floor` 方法可以将一个数舍入到特定的十进制数位。下列语句：

```
y = Math.floor( x * 10 + .5 ) / 10 ;
```

将舍入到  $x$  的十分位（小数点后第一位）。下列语句：

```
y = Math.floor( x * 100 + .5 ) / 100 ;
```

将舍入到  $x$  的百分位（即小数点后的第二位）。编写一个 applet，定义4种方法并使用4种方式舍入  $x$  的数位：

- a) `roundToInteger( number )`
- b) `roundToTenths( number )`
- c) `roundToHundredths( number )`
- d) `roundToThousandths( number )`

对于每个读入的值，应当在运行结果中显示原始值、舍入的整数值、舍入的十分位值、舍入的百分位值以及舍入的千分位值。

4.11 回答下列问题：

- a) 使用“随机”方式选择数字意味着什么？
- b) 为什么 `Math.random` 方法对模拟机会游戏有帮助？
- c) 为什么常常需要对 `Math.random` 方法产生的值进行比例缩放或移位？
- d) 为什么对现实世界进行计算机模拟是一种有用的技术？

4.12 编写语句，在下面的范围内向变量  $n$  赋值随机整数：

- a)  $1 \leq n \leq 2$
- b)  $1 \leq n \leq 100$
- c)  $0 \leq n \leq 9$
- d)  $1\,000 \leq n \leq 1\,112$
- e)  $-1 \leq n \leq 1$
- f)  $-3 \leq n \leq 11$

4.13 对于下列每组整数编写一条语句，从该组中随机打印一个数。

- a) 2, 4, 6, 8, 10
- b) 3, 5, 7, 9, 11
- c) 6, 10, 14, 18, 22

4.14 编写一个方法 `integerPower ( base, exponent )`，返回值为：

$$base^{exponent}$$

例如，`integerPower ( 3, 4 )` 为  $3^4$  ( 或  $3 * 3 * 3 * 3$  )。假定 `exponent` 为正在的非零整数，而 `base` 是一个整数。方法 `integerPower` 应当利用 `for` 或 `while` 结构来控制计算，不要使用任何库方法。将此方法写入一个 applet 中，从用户的键盘上读取 `base` 和 `exponent` 的整数值，使用 `integerPower` 方法完成计算。

4.15 定义一个方法 `hypotenuse`，计算在两条边已知时直角三角形的斜边长，该方法应当接收两个 `double` 类型的参数，并返回 `double` 类型的三角形斜边长度。将此方法写入一个 applet 中，从用户的键盘上读取 `side1` 和 `side2` 的整数值，使用 `hypotenuse` 方法完成计算。确定下列每个直角三角形的斜边长度。

三角形	Side1	Side2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

4.16 编写一个方法 `multiple`，判断一对整数中第二个整数是否是第一个整数的因子。该方法应该接收两个整型的参数，并当第二个整数是第一个的因子时返回 `true`，否则返回 `false`。将此方法写入一个 applet 中，并对输入一系列整数。

4.17 编写一个 applet，输入几个整数，每次向方法 `isEven` 传递一个整数，使用取模运算符判断一个整数是否为偶数。该方法应当接收一个整型参数，在整数为偶数时返回 `true`，否则返回 `false`。

4.18 编写一个方法 `squareOfAsterisks`，显示一个星号组成的方块，方块的边长用整型参数 `side` 指定。例如，如果 `side` 为 4，则该方法显示：

```
* * * *
* * * *
* * * *
* * * *
```

将此方法写入一个 applet，从用户的键盘读取整数值 `side`，使用 `squareOfAsterisks` 方法完



成绘图。注意,这个方法应当从applet的paint方法中调用,并且应该从paint中传入Graphics对象。

- 4.19 修改练习4.18中创建的方法,生成可以使用字符参数fillCharacter包含的任意字符而组成的方块。如side是5且fillCharacter是“#”,那么这个方法应当打印:

```
# # # # #
# # # # #
# # # # #
# # # # #
# # # # #
```

- 4.20 使用与练习4.18和练习4.19中所开发的程序相类似的技术来生成一个程序,可以绘制各种形状。
- 4.21 修改练习4.18中的程序,使用Graphics类的fillRect方法绘制一个实心正方形。方法fillRect接收4个参数: x坐标、y坐标、宽和高,允许用户输入正方形应当出现的坐标(参见练习1.25至练习1.28,以得到关于Graphics类方法的更多信息)。
- 4.22 编写一个程序段完成下列目标:
- 计算整数a除以整数b的商的整数部分。
  - 计算整数a除以整数b的整数余数。
  - 使用a)和b)中开发的程序段编写一个方法displayDigits,接受一个介于1到99999的整数,并打印数字串,每两个数字间隔两个空格。例如,整数4562应当打印成:

```
4   5   6   2
```

- 将c)中开发的方法写入一个applet,从用户的键盘输入一个整数,并通过向方法传入输入的整数和Graphics对象,从applet的paint方法中激活displayDigits。
- 4.23 实现下列整型方法:
- 方法celsius返回一个与华氏温度等价的摄氏温度,使用算式:

$$C = 5.0 / 9.0 * (F - 32);$$

- 方法fahrenheit返回一个与摄氏温度等价的华氏温度,使用算式:

$$F = 9.0 / 5.0 * C + 32;$$

- 使用这些方法编写一个applet,使用户能够输入一个华氏温度并显示相应的摄氏温度,或者输入一个摄氏温度并显示其相应的华氏温度。

- 4.24 编写一个方法minimum3,返回3个浮点数中最小的一个。使用Math.min来实现minimum3。将此方法写入一个applet,读取用户输入的3个值,并使用minimum3确定最小的值。在状态栏中显示结果。
- 4.25 一个整数为完数(perfect number)的条件是,其因子(包括1但不包括该整数本身)的和等于该数。例如,6是一个完数,因为 $6 = 1+2+3$ 。编写一个方法,判断形参number是否为一个完数。在一个applet中应用此方法,判断并打印1到1000之间的完数,打印每个完数的因子。挑战计算机的工作能力,测试一下大于1000的数。
- 4.26 一个整数为质数的条件是它只能被1和自身整除。例如,2、3、5和7都是质数。但4、6、8和9则不是。

- a) 编写一个方法, 判断一个数是否为质数。
  - b) 在一个 applet 中使用此方法, 判断并打印所有 1 到 10 000 之间的质数。在确认已找出了所有的质数之前, 实际测试了这 10 000 个数中的多少个?
  - c) 一开始读者可能认为  $n/2$  次是测试一个数是否为质数的上限, 但实际上最多只需进行  $\sqrt{n}$  次。为什么? 重新编写这个程序, 通过两种方式运行它, 并评价改进的性能。
- 4.27 编写一个方法, 接收一个整数值并返回其逆向数字组成的数。例如, 给定整数 7631, 方法应当返回 1367。将该方法写入一个 applet 中, 从用户处读入一个值, 把运行的结果显示到状态栏上。
- 4.28 两个整数的最大公约数 (GCD) 是两个数能够整除的最大整数。编写一个方法 gcd, 返回两个整数的最大公约数, 将此方法写入一个 applet, 从用户处读入两个值, 在状态栏中显示运行的结果。
- 4.29 编写一个方法 qualityPoints, 输入一个学生的平均成绩, 如果学生的平均成绩为 90 ~ 100 则返回 4, 80 ~ 89 则返回 3, 70 ~ 79 则返回 2, 60 ~ 69 则返回 1, 如果平均成绩低于 60 就返回 0。将此方法编入一个 applet, 从用户处读入一个整数值, 在状态栏上显示该方法的结果。
- 4.30 编写一个 applet 模拟掷硬币。每当用户按下 “Toss” 按钮, 就让程序掷一次硬币, 统计硬币各面出现的次数并显示结果。程序应当调用另一个方法 flip, 该方法没有参数, 并且当硬币为反面时返回 false, 为正面时返回 true。提示: 如果程序正确地模拟了掷硬币, 则硬币每面出现的次数各占一半。
- 4.31 编写一个程序, 帮助小学生学习乘法。利用 Math.random 产生两个一位正整数, 该程序应在状态栏中显示一个如下的问题:

How much is 6 times 7 ?

学生应在文本字段中输入答案。在程序中检查学生的答案, 如果答案正确, 则在 applet 中绘制字符串 “Very good!”, 然后提问另一个乘法问题。如果答案错了, 则在 applet 中绘制字符串 “No. Please try again.”, 然后让学生反复练习同样的问题直到回答正确为止。应当使用一个单独的方法来产生每个新问题。当 applet 开始运行时, 如果每次用户回答正确, 则应调用该方法一次。所有在 applet 中的绘图应当由 paint 方法完成。

- 4.32 计算机在教育中的应用称为计算机辅助教学 (CAI)。在开发 CAI 环境中遇到的一个问题就是学生容易疲劳, 可通过变换计算机的对话来保持学生的注意力, 从而消除疲劳。修改练习 4.31 中的程序, 为每个正确和不正确的答案打印各种评语。对正确答案的评语如下所示:

Very good !  
Excellent !  
Nice work  
Keep up the good work !

对不正确答案的评语如下所示:

No. Please try again.  
Wrong. Try once more.  
Don't give up !  
No. Keep trying.

利用随机数产生器来选择1到4中的一个数,从而给出对于每个答案的一个恰当评语,在 paint 方法中利用一个 switch 结构发出评语

- 4.33 更高级的计算机辅助教学系统可以监控学生在一段时间内的表现:新内容的推出常常基于学生对过去问题的正确回答。修改练习 4.32 的程序,统计学生输入的正确和不正确答案。在学生输入10个答案之后,程序应当计算正确答案的百分比,如果百分比低于75%,则程序应当打印“Please ask your instructor for extra help”并终止运行。
- 4.34 编写一个 applet,按照如下规则玩“猜数”游戏:在程序中,通过选择一个1到1000的整数之间的随机数来确定要猜的数。applet 在一个文本字段旁显示提示:

Guess a number between 1 and 1000

玩家在文本字段中输入第一个数并按下回车键。如果玩家猜错了,程序应当在状态栏中显示“Too high.”、“Try again.”或者“Too low. Try again.”,帮助玩家“接近”正确答案并清除文本字段,以便用户能输入下一个猜测的数。当用户输入了正确答案后,就显示“Congradulations. You guessed the number!”,在状态栏中清除文本字段以便用户可以再次进行游戏。提示:这个问题中使用的查找技术称为二分查找(binary search)。

- 4.35 修改练习 4.34 中的程序以统计玩家已猜测的次数。如果次数是10或更少,就打印“Either you know the secret or you got lucky!”;如果玩家猜测的次数为10次,就打印“Ahah! You know the secret!”;如果玩家猜测多于10次,则打印“You should be able to do better!”。为什么总次数应该少于10次呢?因为使用“理想猜法”的玩家每次能减少一半的数,为什么可以通过10次或更少的次数猜中1到1000之间的任何数?

- 4.36 编写一个递归方法 power( base, exponent ),调用时返回:

$base^{exponent}$

例如,  $power(3,4) = 3 * 3 * 3 * 3$ 。假定 exponent 是一个大于等于1的整数。提示:递归步骤应使用下列关系:

$base^{exponent} = base \cdot base^{exponent-1}$

且终止条件在 exponent 等于1时发生,因为:

$base^1 = base$

将此方法写入一个 applet,使用户可以输入 base 和 exponent。

- 4.37 斐波纳契数列 0, 1, 1, 2, 3, 5, 8, 13, 21, ... 以0和1开始,且其特征是后续项为前两项之和。
- 编写一个非递归方法 fibonacci(n), 计算 n 个斐波纳契数。将此方法写入一个 applet, 使用户可以输入 n 的值。
  - 确定可以在系统中计算出的最大的斐波纳契数。修改 a) 部分, 使用 double 代替 int 类型, 计算并返回斐波纳契数, 使用修改过的程序重新编写 b) 部分。
- 4.38 (汉诺塔) 每一位计算机科学家都必须掌握一些经典问题, 汉诺塔 (如图 4.19 所示) 就是其中之一。这个故事发生在远东的一所寺庙中, 僧侣们试图将一堆圆盘从一个柱子移到另一个柱子上面。最初的64个圆盘串在一个柱子上, 并且从底部向上逐渐减小。僧侣们正试图将这些圆盘从一个柱子移到另一个的上面, 但必须遵守规则, 即一次只能移动一只圆盘, 且在任何时刻大圆盘不能在小圆盘之上, 第三个柱子可用来临时放置圆盘。

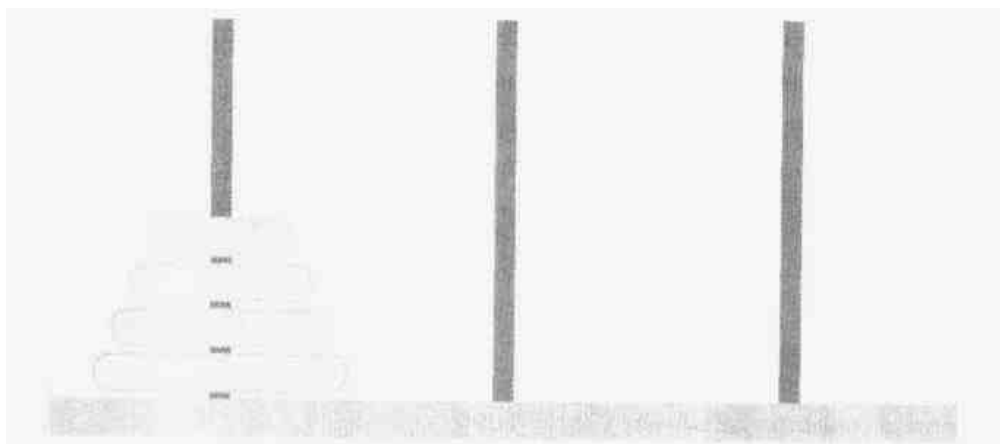


图 4.19 有 4 只圆盘的汉诺塔

假定僧侣们正试图将所有的圆盘从 1 号柱子移往 3 号柱子。我们希望开发一个算法，打印出从柱子到柱子之间移动圆盘的精确顺序。如果使用传统方法解决这一问题，我们很快就会发现这个问题十分复杂。相反，如果使用递归方法解决这个问题，则该问题立即变得易于处理。移动  $n$  个圆盘可以看成如下的只移动  $n-1$  只圆盘（因此是递归）的过程：

- a) 从柱子 1 移动  $n-1$  只圆盘到柱子 2，利用柱子 3 作为临时存放区。
- b) 从柱子 1 移动最后一只圆盘（最大的一只）到柱子 3。
- c) 从柱子 2 移动  $n-1$  只圆盘到柱子 3，利用柱子 1 作为临时存放区。

该过程在移动  $n=1$  只圆盘时结束，即达到基本情况，这时直接移动圆盘而无需一个临时存放区。编写一个 applet 解决汉诺塔问题，使用一个带有 4 个参数的递归 `tower` 方法，其中包括：

- a) 要移动的圆盘数
- b) 这些圆盘最初所在的柱子
- c) 这些圆盘要移到的柱子
- d) 作为临时存放区的柱子

程序应当准确打印指令，它将圆盘从原柱子移向目标柱子。例如，为了把一堆圆盘从柱子 1 移到柱子 3，应当打印下面的移动序列：

1 → 3（这是指一只圆盘从柱子 1 移到柱子 3）

1 → 2

3 → 2

1 → 3

2 → 1

2 → 3

1 → 3

提示：`tower` 方法必须在函数首部上带有关键字 `static`，我们将在第 6 章详细解释。

- 4.39 任何可用递归实现的程序也可以使用迭代来实现，尽管有时会更困难且不太清晰。试着编写一个汉诺塔程序的迭代版本，把迭代版本同练习 4.38 的递归版本相比较。考察一下性能问题、清晰性以及程序的正确性。
- 4.40（可视化递归）观察递归“动作”是一件有意思的事。修改图 4.12 的阶乘方法，打印其局部变量和递归调用的参数。对于每次递归调用，在单独一行上显示输出并添加一个缩

进标志。尽可能使输出清晰、有趣且有意义。我们的目的是设计和实现一个输出格式，帮助他人更好地了解递归。

- 4.41 整数  $x$  和  $y$  的最大公约数  $x$  和  $y$  是能整除的最大整数。编写一个递归方法 `ged`，返回  $x$  和  $y$  的最大公约数。 $x$  和  $y$  的 `ged` 递归定义如下：如果  $y$  为 0，则 `ged(x, y)` 为  $x$ ；否则 `ged(x, y)` 为 `ged(y, x % y)`，其中 `%` 为取模运算符。使用这个方法替换在练习 4.28 中编写的 `applet`。
- 4.42 练习 4.31 至练习 4.33 开发了一个计算机辅助教学程序，教授小学生学习乘法。本练习建议增强该程序
- a) 修改程序，允许用户输入等级。等级 1 说明只在问题中使用一位数字，等级 2 表示可以使用两位数字，依次类推。
  - b) 修改程序，允许用户挑选他想研究的算术问题。选择 1 说明只有加法，选择 2 说明只有减法，选择 3 说明只有乘法，选择 4 说明只有除法，选择 5 说明所有这些类型的随机复合问题。
- 4.43 编写方法 `distance`，计算  $(x1, y1)$  和  $(x2, y2)$  两点间的距离，所有的数字和返回值均为 `double` 类型。将此方法写入一个 `applet`，使用户能输入点的坐标。
- 4.44 下面的程序完成了什么操作？

```
// Parameter b must be a positive
// integer to prevent infinite recursion
int mystery ( int a, int b )
{
    if( b == 1 )
        return a ;
    else
        return a + mystery( a, b-1 ) ;
}
```

- 4.45 修改练习 4.44 中的方法，使之在取消第二个参数为非负的限制后能正确操作。然后将此方法写入一个 `applet`，使用户能输入两个整数并测试此方法。
- 4.46 编写一个应用程序，尽可能多地测试图 4.2 中的 `Math` 类的方法。让程序对于广泛的参数值打印出返回值表，以便于观察每个方法。
- 4.47 找出下面递归方法中的错误并说明如何修改它：

```
int sum ( int n )
{
    if( n == 0 )
        return 0 ;
    else
        return n + sum( n ) ;
}
```

- 4.48 修改图 4.9 中的掷骰子程序，允许加入赌注。初始化变量 `bankBalance`（余额）为 \$1 000，提示玩家输入一个 `wager`（赌注），检查 `wager` 是否小于等于 `bankBalance`，如果不是，则让用户重新输入 `wager`，直至输入了一个合法的 `wager`。在一个正确的 `wager` 输入后，运行一次掷骰子游戏。如果用户赢了，则把 `wager` 增加到 `bankBalance`，打印新的 `bankBalance`。

如果用户输了,则在bankBalance中减去wager,打印出新的bankBalance,检查bankBalance是否变成零,如果是就打印消息“Sorry you busted!” 在游戏进行时,打印各种聊天式的消息,如“ Oh, you're going for broke, huh?”,或者“Aw cmon, take a chance!”,或者“You're up big, Now's the time to cash in your chips!”。

- 4.49 编写一个 applet, 利用 circleArea 方法提示用户输入一个圆的半径, 计算并打印该圆的面积。

# 第5章 数 组

## 教学目标

- 介绍数组数据结构
- 理解使用数组进行存储、排序，以及查找列表和表格中的值
- 理解如何声明数组、初始化数组和引用数组中的单个元素
- 学会向方法传递数组
- 理解基本的排序方法
- 学会声明和使用多维数组

## 5.1 简介

本章将引入数据结构这个重要概念。数组是包含相同类型的相关数据项的数据结构。数组是“静态”的实体，这是因为数组创建后的大小将保持不变，不过一个数组引用可以重新赋值为一个新的、不同大小的数组。在第17章中，我们将介绍一些可随程序执行而增长和缩减的动态数据结构，例如链表、队列、堆栈以及树。在第18章中，我们将讨论Vector类，这是一个类似数组的类，其对象大小可以随Java程序不断改变存储需求而增长或缩减。

## 5.2 数组

一个数组是一组连续的存储单元，其每个成员均有相同的名称和类型。为了引用数组中的元素，我们定义了数组名以及数组中元素的位置序号。

图5.1展示了一个名为c的整数数组，该数组包含了12个元素。可以使用数组名后带有方括号([])括起来的位置序号来引用数组中的某一特定元素。元素位置的计数从0开始。因此，数组c的第1个元素记为c[0]，数组c的第2个元素记为c[1]，数组c的第7个元素记为c[6]，依次类推，数组c的第i个元素记为c[i-1]。数组命名的规则同其他变量命名一样。方括号中的位置序号称为下标，下标必须是整数或整数表达式。如果程序中使用整数表达式作为下标，那么先要计算该表达式才能确定这个下标的值。例如，如果我们假定变量a等于5，变量b等于6，那么下列语句：

```
c[ a + b ] += 2;
```

就是给元素c[11]加2。注意，带下标的数组名是一个左值(lvalue)，也就是它可以放在赋值语句的左边，以便将一个新值放入数组元素中。

数组名（注意：该数组的所有元素都有相同的名字）

c[ 0 ]	-45
c[ 1 ]	6
c[ 2 ]	0
c[ 3 ]	72
c[ 4 ]	1543
c[ 5 ]	-89
c[ 6 ]	0
c[ 7 ]	62
c[ 8 ]	-3
c[ 9 ]	1
c[ 10 ]	6453
c[ 11 ]	78

↑  
数组c内元素的位置序号

图 5.1 一个包含 12 个元素的数组

让我们仔细研究图 5.1 中的数组 `c`。数组的名字是 `c`，Java 中每个数组的长度都保存在变量 `length` 中，表达式 `c.length` 指定了数组的长度。数组的 12 个元素分别记为 `c[0]`，`c[1]`，`c[2]`， $\dots$ ，`c[11]`。`c[0]` 的值是 -45，`c[1]` 的值是 6，`c[2]` 的值是 0，`c[7]` 的值是 62，`c[11]` 的值是 78。我们可用下面的语句计算出数组 `c` 中前三个元素值的总和，并将结果存入变量 `sum` 中：

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

下面的语句是将第 7 个元素除以 2，并将结果赋给变量 `x`：

```
x = c[ 6 ] / 2;
```

#### 常见编程错误 5.1

注意，“数组的第 7 个元素”和“数组元素 7”之间是有差别的。由于数组下标自 0 开始，所以“数组的第 7 个元素”的下标是 6，而“数组元素 7”的下标为 7，实际上是该数组的第 8 个元素。这一混淆是产生“差 1 错误”（off-by-one）的根源。

在 Java 中，用于括住数组下标的方括号是一个运算符，方括号和圆括号有着相同的优先级。图 5.2 列出了本书中各运算符的优先级和结合律。这些运算符自顶向下按优先级递减的顺序排列，请参看附录 A 以得到完整的运算符优先级表。

运算符	结合律	类型
() [] .	从左向右	括号（优先级最高）
++ --	从右向左	一元后缀
++ -- + - !(type)	从右向左	一元运算符
* / %	从左向右	乘法
+ -	从左向右	加法



(续表)

运算符	结合律	类型
< <= > >=	从左向右	关系
== !=	从左向右	相等关系
&	从左向右	布尔逻辑与
^	从左向右	布尔逻辑异或
	从左向右	布尔逻辑或
&&	从左向右	逻辑与
	从左向右	逻辑或
?:	从右向左	条件
= += -= *= /= %=	从右向左	赋值

图 5.2 运算符的优先级和结合律

### 5.3 声明数组和分配数组

数组要占据存储空间。在Java中,使用new运算符来动态分配对象。声明数组时,程序员先定义元素的类型,然后使用new运算符分配数组所需的空间。为了分配图5.1中整数数组c的12个元素,可以采用以下声明:

```
int c[] = new int[ 12 ];
```

上面的语句也可以分成两步来执行:

```
int c[];           //declares the array
c = new int[ 12 ]; //allocates the array
```

为数组分配空间以后,其元素将自动初始化。如果初始化为基本数据类型,则为0值;布尔类型则为false;引用类型(以及其他非基本类型)则为null(空)。

#### 常见编程错误 5.2

与在其他编程语言(例如C或C++)中声明数组不同,Java数组的元素个数不能在数组名后的方括号中定义,否则将导致语法错误。例如,声明“int c[ 12];”会产生语法错误。

可以在一条声明语句中为多个数组分配空间。下面的声明为数组b分配了100个元素,为数组x分配了27个元素:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

声明数组时,可以将数组的类型和方括号放在声明语句的开头,以指出数组声明中的所有标识符,例如:

```
double[] array1,array2;
```

这条语句中array1和array2都是double类型数组,如前所述,可以在声明语句中声明和初始化数组,下面的语句为array1分配了10个元素,为array2分配了20个元素:

```
double[] array1 = new double[ 10 ],
        array2 = new double[ 20 ];
```

可以声明任何类型的数组。关键是记住基础数据类型的数组中,元素包含所声明数据类型的

值。例如，int 数组中的元素包含 int 类型的值。但是，在基础类型的数组中，元素则是指向该数组数据类型的对象的引用。例如，String 数组中的元素为指向 String 的引用，在数组中保存引用，引用默认为 null 值。

## 5.4 使用数组的实例

这一节将给出几个使用数组的例子，演示了声明数组、分配数组、初始化数组以及操纵数组元素的各种方法。为了简单起见，这一节的数组都为 int 类型，但实际上程序可以声明任何类型的数组。

### 5.4.1 分配数组并初始化数组元素

图 5.3 中的程序使用 new 运算符动态分配了一个具有 10 个元素的数组，将该数组初始化为 0，然后以表格形式将其打印出来。

```
1 // Fig. 5.3: IntArray.java
2 // initializing an array
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class IntArray extends Applet {
7     int n[]; // declare an array of integers
8
9     // initialize instance variables
10    public void init()
11    {
12        n = new int [ 10 ]; // dynamically allocate array
13    }
14
15    // paint the applet
16    public void paint( Graphics g )
17    {
18        int yPosition = 25; // starting y position on applet
19
20        g.drawString( "Element", 25, yPosition );
21        g.drawString( "Value", 100, yPosition );
22
23        for ( int i = 0; i < n.length; i++ ) {
24            yPosition += 15;
25            g.drawString( String.valueOf( i ), 25, yPosition );
26            g.drawString( String.valueOf( n[ i ] ),
27                        100, yPosition );
28        }
29    }
30 }
```

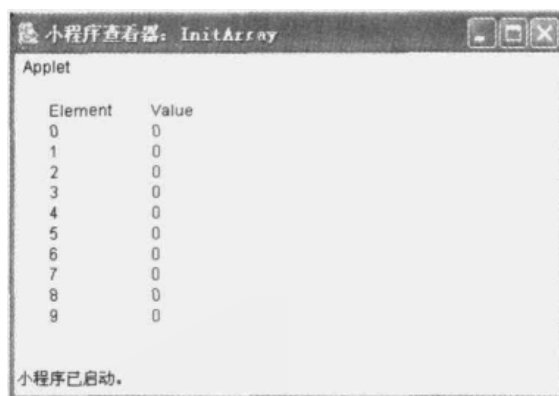


图 5.3 将数组中的元素初始化为 0

在图 5.3 中, `paint` 方法的前两条语句是在 `for` 结构中打印各列的列标题。变量 `yPosition` 用于确定 `drawString` 方法在输出结果中的位置。`String.valueOf` 方法用于将每个整数转换成可以在 applet 中显示的字符串, `for` 结构中的方法 `n.length` (第 23 行) 用于确定该数组的长度。

数组中的元素可在数组声明中初始化, 这只需在声明之后加上一个等号以及一个由逗号分隔的初始化值 (initializer) 列表 (用花括号括起) 即可。在这种情况下, 数组的大小由初始化值列表中的元素个数确定。例如下面的语句:

```
int n[ ] = { 1, 2, 3, 4, 5 };
```

这样就创建了一个 5 元素数组。

#### 5.4.2 使用初始化值列表来初始化数组元素

图 5.4 中的程序使用 10 个数值 (第 7 行) 初始化了一个整型数组, 并以表格形式将其打印出来。

```

1      // Fig. 5.4: InitArray.java
2      // initializing an array with a declaration
3      import java.awt.Graphics;
4      import java.applet.Applet;
5
6      public class InitArray extends Applet {
7          int n[ ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
8
9          // paint the applet
10         public void paint( Graphics g )
11         {
12             int yPosition = 25;    // starting y position on applet
13
14             g.drawString( "Element", 25, yPosition );
15             g.drawString( "Value", 100, yPosition );
16
17             for ( int i = 0; i < n.length; i++ ) {
18                 yPosition += 15;
19                 g.drawString( String.valueOf( i ), 25, yPosition );
20                 g.drawString( String.valueOf( n[ i ] ),
21                             100, yPosition );
22             }
23         }
24     }
```



图 5.4 在声明中初始化数组的元素

### 5.4.3 计算存储在数组元素中的值

图 5.5 中的程序将一个 10 元素数组的元素初始化为整数 2、4、6、…、20，然后将其以表格形式打印出来。这些数据是由循环计数器的值乘 2 再加上 2 得到的。

其中的第 7 行：

```
final int arraySize = 10;
```

使用 `final` 修饰符声明了一个所谓的常量变量 `arraySize`，其值为 10。常量变量必须用一个常量表达式初始化，而且在声明之后就不能再修改了（参见图 5.6 及图 5.7）。如果试图在声明常量变量之后对其加以修改，编译器就会产生如下的出错消息：

```
Can't assign a value to a final variable
```

```

1  // Fig. 5.5: InitArray.java
2  // initialize array s to the even integers from 2 to 20
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class InitArray extends Applet {
7      final int arraySize = 10;
8      int s [ ];
9
10     // initialize instance variables
11     public void init()
12     {
13         s = new int [ arraySize ];
14
15         // Set the values in the array
16         for ( int i = 0; i < s.length; i++ )
17             s[ i ] = 2 + 2 * i;
18     }
19
20     // paint the applet
21     public void paint( Graphics g )
22     {
23         int yPosition = 25;    // starting y position on applet
24
25         g.drawString( "Element", 25, yPosition );

```

```

26         g.drawString( "Value", 100, yPosition );
27
28         for ( int i = 0; i < s.length; i++ ) {
29             yPosition += 15;
30             g.drawString( String.valueOf( i ), 25, yPosition );
31             g.drawString( String.valueOf( s[ i ] ),
32                         100, yPosition );
33         }
34     }
35 }

```

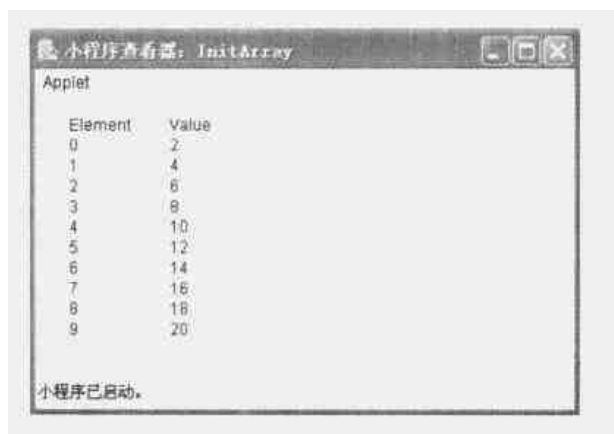


图 5.5 生成数组元素的值

```

1 // Fig. 5.6: FinalTest.java
2 // a final object must be initialized
3 import java.applet.Applet;
4
5 public class FinalTest extends Applet {
6     final int x; // Error: Final variables must be initialized
7 }

```

**输出结果:**

```

FinalTest.java:7:Final variables must be initialized:int x
    final int x; // Final variables must be initialized
        ^
1 error

```

图 5.6 必须初始化 final 对象

```

1 // Fig. 5.7: FinalTest.java
2 // using a properly initialized constant variable
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class FinalTest extends Applet {
7     final int x = 7; // initialize constant variable
8
9     public void paint( Graphics g )
10    {
11        g.drawString( "The value of x is: " + x, 25, 25 );
12    }
13 }

```

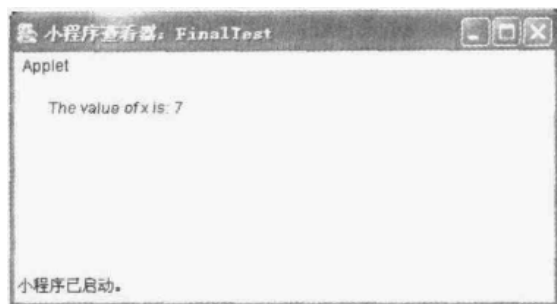


图 5.7 正确地初始化并使用常量变量

常量变量也称为命名常量或只读变量，它们常常用于增加程序的可读性。

#### 常见编程错误 5.3

在一条可执行的语句中，向一个常量变量赋值是一个语法错误。图 5.8 中的程序将整数数组 *a* 中各元素的值求和，for 循环体中的语句完成了求和过程。记住，数组 *a* 的初始化值将读入程序。例如，用户可以通过 applet 中的文本字段输入数值。

#### 5.4.4 对数组元素求和

下面的例子（如图 5.8 所示）将汇总一次调查中收集的数据结果，该问题如下所示：

要求 40 名学生使用一个 1 到 10（1 代表最糟，10 代表最好）的标准给学生餐厅的餐饮质量分级。将 40 个回答放入一个整型数组中，并对该调查的结果求和。

```

1    // Fig. 5.8: SumArray.java
2    // Compute the sum of the elements of the array
3    import java.awt.Graphics;
4    import java.applet.Applet;
5
6    public class SumArray extends Applet {
7        int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
8        int total;
9
10       // initialize instance variables
11       public void init()
12       {
13           total = 0;
14
15           for ( int i = 0; i < a.length; i++ )
16               total += a[ i ];
17       }
18
19       // paint the applet
20       public void paint( Graphics g )
21       {
22           g.drawString( "Total of array elements: " + total,
23                       25, 25 );
24       }
25     }

```

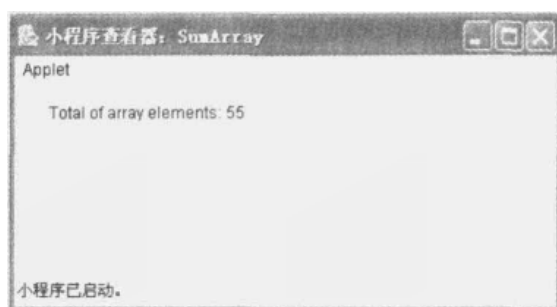


图 5.8 计算数组元素之和

### 5.4.5 使用数组分析调查结果

下面，我们给出一个典型的数组处理应用程序（如图 5.9 所示）。我们求出每种回答（1 到 10）的个数和。数组 `responses` 是一个 40 元素的整型数组，其中保存了学生对调查所做的回答。我们使用一个 11 元素的数组 `frequency` 来计算每种回答出现的次数。忽略第一个元素 `frequency[0]`，这是因为使用 `frequency[1]` 来反映第一个回答结果要比使用 `frequency[0]` 更有逻辑性。

#### 编程技巧 5.1

尽量增加程序的清晰度。有时牺牲有效的内存使用和处理器时间来换取更清晰的程序是值得的。

#### 性能提示 5.1

有时候对程序性能的要求要远大于对程序清晰度的要求。

```

1    // Fig. 5.9: StudentPoll.java
2    // Student poll program
3    import java.awt.Graphics;
4    import java.applet.Applet;
5
6    public class StudentPoll extends Applet {
7        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
8                           1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
9                           6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
10                          5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
11        int frequency[];
12
13        // initialize instance variables
14        public void init()
15        {
16            frequency = new int[ 11 ];
17
18            for ( int answer = 0; answer < responses.length; answer++ )
19                ++frequency[ responses[ answer ] ];
20        }
21
22        // paint the applet
23        public void paint( Graphics g )
24        {
25            int yPosition = 25;    // starting y position on applet
26
27            g.drawString( "Rating", 25, yPosition );
28            g.drawString( "Frequency", 100, yPosition );
29        }

```

```

30         for ( int rating = 1;
31             rating < frequency.length; rating++ ) {
32             yPosition += 15;
33             g.drawString( String.valueOf( rating ),
34                           25, yPosition );
35             g.drawString( String.valueOf( frequency[ rating ] ),
36                           100, yPosition );
37         }
38     }
39 }

```

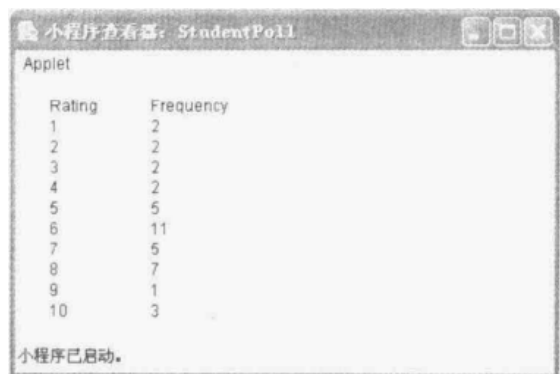


图 5.9 一个简单的学生调查分析程序

第 18 行中的 for 循环每次从数组 `responses` 中取出一个回答结果，接着将 `frequency` 数组的 10 个计数器（`frequency[ 1 ]` 到 `frequency[ 10 ]`）中的一个加 1。循环中的关键语句是：

```
++frequency [ responses [ answer ] ] ;
```

该语句根据 `responses[ answer ]` 的值增加相应的 `frequency` 计数器的值。

让我们考虑这个 for 循环的几次迭代。当计数器 `answer` 为 0 时，`responses[ answer ]` 为 1，于是“`++frequency[ responses[ answer ] ]`”实际上解释为：

```
++frequency [ 1 ] ;
```

当计数器 `answer` 为 1 时，`responses[ answer ]` 为 2，于是语句“`++frequency[ responses[ answer ] ]`”实际上解释为：

```
++frequency [ 2 ] ;
```

它增加了元素 2 的值。

当计数器 `answer` 为 2 时，`responses[ answer ]` 为 6，于是语句“`++frequency[ responses[ answer ] ]`”实际上解释为：

```
++frequency [ 6 ] ;
```

它增加了元素 6 的值，依次类推。注意在本例中不考虑在此次调查中处理的回答数，只需要一个 11 元素的数组来汇总结果。该程序的运行结果是正确的，因为当用 `new` 声明数组 `frequency` 时，数组的元素都将自动初始化为 0。如果各元素没有初始化为 0，那么调查中每种回答的频率计数器就不会得到正确的值。

如果数据包括 13 这样的非法值，程序就会试图将 `frequency[ 13 ]` 加 1，这就超出了数组的边界。在 C 和 C++ 语言中，编译和执行过程中都允许出现这种引用。程序将越过数组的末尾，指向它认定的元素序号为 13 的分配空间，然后直接向内存中的那个位置加 1。这样可能会在无意中修改其他的



程序变量,甚至导致其他程序的运行失败

Java 提供了防止越界访问数组元素的机制

#### 常见的编程错误 5.4

超越数组边界来引用元素是一个逻辑错误

#### 测试与调试提示 5.1

当编译Java程序时,编译器检查数组元素的引用以确保它们是合法的(例如,所有的下标必须大于等于0并小于数组长度)。如果在执行时对数组元素进行了非法引用,Java就会产生一个异常

#### 测试与调试提示 5.2

异常用于指明在一个程序中发生的错误,它们可使程序员从错误中恢复并继续执行程序,而不是非正常地终止程序。当出现一个非法数组引用时,就会产生一个ArrayIndexOutOfBoundsException异常,详细内容请参见第14章。

#### 测试与调试提示 5.3

当循环一个数组时,其数组下标绝不能小于0,并且要小于数组中全部元素的个数(即小于数组的大小)。要确保循环终止条件不会导致引用超出这一范围的元素。

#### 测试与调试提示 5.4

程序应当能够确认所有输入值的正确性,防止错误信息影响程序的计算。

下面的例子(如图5.10所示)从数组中读取数据然后将信息以条形图或柱形图的形式绘出,即每打印一个数,旁边再打印包含相应星号数的横条。嵌套的for循环(第26行)用于绘出这些横条。

```
1 // Fig. 5.10: Histogram.java
2 // Histogram printing program
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class Histogram extends Applet {
7     int n[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
8
9     // paint the applet
10    public void paint( Graphics g )
11    {
12        int xPosition;           // position of * in histogram
13        int yPosition = 25;      // vertical position in applet
14
15        g.drawString( "Element", 25, yPosition );
16        g.drawString( "Value", 100, yPosition );
17        g.drawString( "Histogram", 175, yPosition );
18
19        for ( int i = 0; i < n.length; i++ ) {
20            yPosition += 15;
21            g.drawString( String.valueOf( i ), 25, yPosition );
22            g.drawString( String.valueOf( n[ i ] ),
23                        100, yPosition );
24            xPosition = 175;
25
26            for ( int j = 1; j <= n[ i ]; j++ ) { // print one bar
27                g.drawString( "*", xPosition, yPosition );
```

```

28         xPosition += 7;
29     }
30 }
31 }
32 }

```

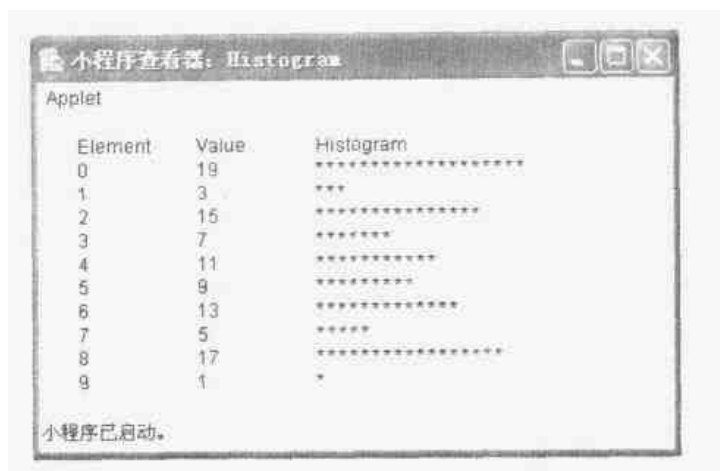


图 5.10 一个打印柱形图的程序

下面，将以数组的方法实现第4章曾介绍过的掷骰子程序。该问题是将一个6面的骰子掷6000次，以测试随机数产生器是否生成了随机数，如图 5.11 所示。

```

1  // Fig. 5.11: RollDie.java
2  // Roll a six-sided die 6000 times
3  import java.awt.Graphics;
4  import java.applet.Applet;
5  import java.util.Random;
6
7  public class RollDie extends Applet {
8      int face;
9      int frequency [ ];
10     Random r;    // create the random number generator
11
12     // initialize instance variables
13     public void init()
14     {
15         frequency = new int [ 7 ];
16         r = new Random();
17
18         for ( int roll = 1; roll <= 6000; roll++ ) {
19             face = 1 + Math.abs( r.nextInt() % 6 );
20             ++frequency[ face ];
21         }
22     }
23
24     // paint the applet
25     public void paint( Graphics g )
26     {
27         int yPosition = 25;
28
29         g.drawString( "Face", 25, yPosition );
30         g.drawString( "Frequency", 100, yPosition );

```

```
31
32     for ( face = 1; face < frequency.length; face++ ) {
33         yPosition += 15;
34         g.drawString( String.valueOf( face ), 25, yPosition );
35         g.drawString( String.valueOf( frequency[ face ] ),
36                     100, yPosition );
37     }
38 }
39 }
```

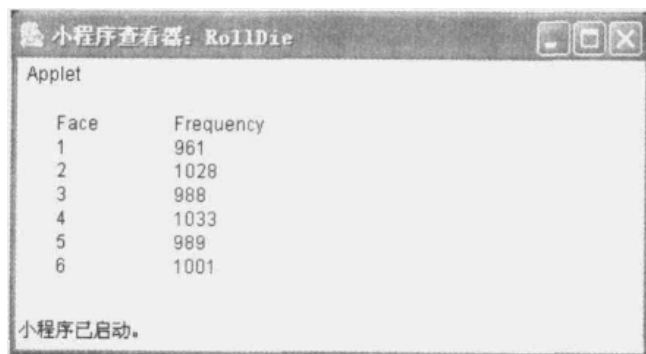


图 5.11 使用数组实现的掷骰子程序

## 5.5 引用和引用参数

在许多编程语言中,有两种向方法(或函数)传递参数的方法,分别称为按值调用和按引用调用。

正常情况下由程序员自己确定要传递的方式。当一个参数通过按值调用方式传递时,就会产生该参数值的一个副本,并将其传递给被调用的方法。

### 测试与调试提示 5.5

使用按值调用时,对被调用方法副本值的改变不会影响到调用方法的原参数的值,这样就防止偶然的副作用严重影响到开发正确和可靠的软件系统。

当按引用调用传递参数时,调用者赋予被调用的方法直接访问调用者数据的能力,被调用方法甚至还可以修改此数据。按引用调用由于消除了复制大量数据的任务,因此能够提高程序性能,但由于被调用方法能够访问调用者的数据,因此可能降低了安全性。

### 软件工程视点 5.1

Java 不像其他语言,它不允许程序员选择是按值调用还是按引用调用来传递每个参数。基本数据类型变量通常按值调用传递,而对象则通过按引用调用传递。引用本身是通过按值调用传递的,即将引用的副本传递给方法。当方法接收一个对象的引用时,可以直接处理该对象。

### 软件工程视点 5.2

当使用 return 语句从一个方法中返回信息时,基本数据类型的变量通常按值返回,对象通常按引用返回(即返回对象的引用)。

为了使用按引用调用传递一个对象,只需在调用该对象引用的方法中用名字定义即可。在被调用方法中通过其参数名指出的对象,实际上就是内存中的原对象,该对象可以由被调用方法直接访问。

由于在 Java 中数组也被视为对象,因此数组是通过按引用调用方式传递的,即一个被调用的

方法能够访问调用者的原始数组中的元素。一个数组的名字实际上就是对一个对象的引用,对象包含数组元素以及length实例变量,用以指明数组中元素的个数。在下一节,我们将通过数组来说明按值调用和按引用调用的用法。

#### 性能提示 5.2

按引用方式传递数组是出于对性能的考虑。如果数组按值传递,那么传递的将是每个元素的副本。对于大型的、频繁传递的数组而言,这是很浪费处理时间的,而且会由于数组的副本而占用可观的存储空间。

## 5.6 向方法传递数组

向一个方法传递数组参数时,数组的名字不要带方括号。例如,如果数组hourlyTemperatures已经声明为:

```
int hourlyTemperatures[] = new int[ 24 ];
```

则方法调用:

```
modifyArray( hourlyTemperatures );
```

将数组hourlyTemperatures传递给方法modifyArray。在Java中,每个数组对象的大小都是“已知”的(通过length实例变量给出)。因此,当我们向一个方法传递数组对象时,并不需要将数组大小作为一个参数单独传递。

尽管整个数组以及非基本类型的单个元素引用的对象是按引用调用传递的,但基本类型的单个数组元素仍按值调用传递,这与简单变量完全相同。这些简单的单个数据称为标量(scalar)或标量数(scalar quantity)。为了向方法传递一个数组元素,可以在方法调用中使用该数组元素的下标名作为参数。

对于通过方法调用来接收数组的方法而言,该方法的参数表中必须指明要接收的数组。例如,方法modifyArray的首部应当写成:

```
void modifyArray( int b[] )
```

以指明modifyArray将通过参数b接收一个整数数组。由于数组是通过引用传递,当被调用的方法使用数组名b时,实际上指的是调用一方中的实际数组(即在前面调用中的hourlyTemperatures数组)。

图5.12中的程序展示了传递整个数组和一个数组元素的不同。该程序首先打印了整型数组a中的5个元素。接着,将a传递到方法modifyArray中,在那里将a的每个元素乘以2。然后在paint方法中打印a。正如输出显示的那样,a的元素的确由modifyArray方法所修改。接着程序打印a[3]的值并将其传递到方法modifyElement,该方法将其参数乘以2。注意,当a[3]在paint方法中打印时,由于单个基本类型的数组元素是按值调用,所以始终未被修改。

```
1 // Fig. 5.12: PassArray.java
2 // Passing arrays and individual array elements to methods
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class PassArray extends Applet {
7     int a[] = {0, 1, 2, 3, 4};
8
9     public void paint( Graphics g )
```

```

10     {
11         int xPositon = 20, yPositon = 25;
12
13         g.drawString(
14             "Effects of passing entire array call-by-reference:",
15             xPositon, yPositon );
16         yPositon += 15;
17         g.drawString( "The values of the original array are:",
18             xPositon, yPositon );
19         xPositon += 10;
20         yPositon += 10;
21
22         for ( int i = 0; i < a.length; i++ ) {
23             g.drawString( String.valueOf( a[ i ] ),
24                 xPositon, yPositon );
25             xPositon += 10;
26
27
28             xPositon = 20;
29             yPositon += 30;
30
31             modifyArray( a ); // array a passed call-by-reference
32
33             g.drawString( "The values of the modified array are:",
34                 xPositon, yPositon );
35             xPositon += 15;
36             yPositon += 15;
37
38             for ( int i = 0; i < a.length; i++ ) {
39                 g.drawString( String.valueOf( a[ i ] ),
40                     xPositon, yPositon );
41                 xPositon += 15;
42             }
43
44             xPositon = 25;
45             yPositon += 30;
46
47             g.drawString(
48                 "Effects of passing array element call-by-value:",
49                 xPositon, yPositon );
50             yPositon += 15;
51             g.drawString( "a[3] before modifyElement: " + a[ 3 ],
52                 xPositon, yPositon );
53             yPositon += 15;
54
55             modifyElement( a[ 3 ] );
56
57             g.drawString( "a[3] after modifyElement: " + a[ 3 ],
58                 xPositon, yPositon );
59         }
60
61         public void modifyArray( int b[ ] )
62         {
63             for ( int j = 0; j < b.length; j++ )
64                 b[ j ] *= 2;
65         }

```

```

66
67     public void modifyElement( int e )
68     {
69         e *= 2;
70     }
71 }

```

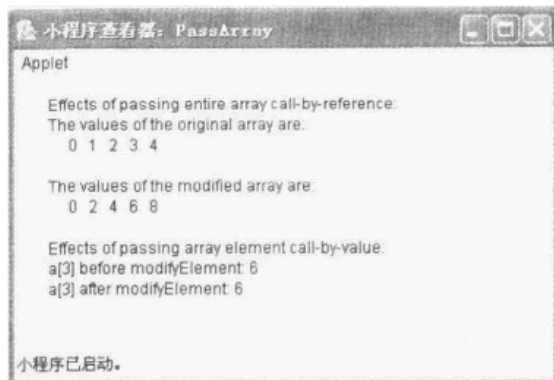


图 5.12 向方法传递数组和单个数组元素

## 5.7 数组排序

数据排序(比如将数据排列成某种特定的顺序,这种顺序可能是升序或降序)是最重要的计算应用之一。银行根据账号余额对所有的支票进行排序,以便能够准备好在每个月末发布的报告。电话公司根据姓氏以及名字对其账户进行排序,以便查找电话号码。可以想像,每个组织都必须将某种数据排序,而在很多情况下,这些数据的数量可能是巨大的。数据排序是一个具有挑战性的问题,排序问题已经吸引了很多的研究力量投入到这个领域中,本章我们只讨论最简单的排序方法。在本章练习和第 17 章中,我们将考察具有更高性能的复杂排序方法。

### 性能提示 5.3

通常,最简单的算法的性能较差,但其优点是易于编写、测试和调试。为了实现更好的性能,通常需要更复杂的算法。

图 5.13 中的程序将 10 元素数组 `a` 中的值按升序排列。我们所用到的技术称为冒泡排序(bubble sort)或下沉排序(sinking sort),这是因为较小的值逐渐“冒泡”至数组的顶端,就像水中上升的气泡一样,同时较大的值沉向数组底端。这个算法将对整个数组进行数次遍历,在每次遍历时,逐一比较相邻的元素。如果一对元素已经是升序的(或者相等),就保持其位置;如果一对元素是降序的,则将它们交换。注意,在 `print` 方法中(第 31 行定义)输出数组内容之前先输出一个字符串。

```

1 // Fig. 5.13: BubbleSort.java
2 // This program sorts an array's values into
3 // ascending order
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class BubbleSort extends Applet {
8     int a[] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
9     int hold; // temporary holding area for swap

```

```

10
11     public void paint( Graphics g )
12     {
13         print( g, "Data items in original order", a, 25, 25 );
14
15         sort();
16
17         print( g, "Data items in ascending order", a, 25, 55 );
18     }
19
20     public void sort()
21     {
22         for ( int pass = 1; pass < a.length; pass++ ) // passes
23             for ( int i = 0; i < a.length - 1; i++ ) // one pass
24                 if ( a[ i ] > a[ i + 1 ] ) {           // one comparison
25                     hold = a[ i ];                     // one swap
26                     a[ i ] = a[ i + 1 ];
27                     a[ i + 1 ] = hold;
28                 }
29     }
30
31     public void print( Graphics g, String head, int b[ ],
32                       int x, int y )
33     {
34         g.drawString( head, x, y );
35         x += 15;
36         y += 15;
37
38         for ( int i = 0; i < b.length; i++ ) {
39             g.drawString( String.valueOf( b[ i ] ), x, y );
40             x += 20;
41         }
42     }
43 }

```

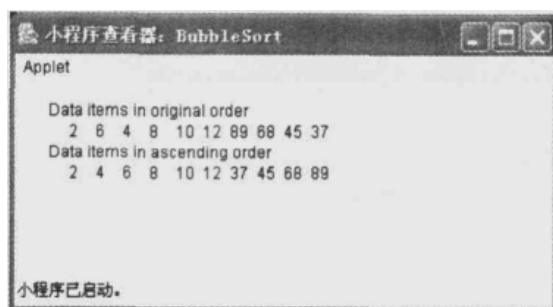


图 5.13 使用冒泡排序法对数组排序

程序首先比较  $a[0]$  和  $a[1]$ ，然后是  $a[1]$  和  $a[2]$ ，接下来是  $a[2]$  和  $a[3]$ ，如此继续下去，直到比较完  $a[8]$  和  $a[9]$ ，才完成此次遍历。尽管数组中有 10 个元素，但只比较 9 次。由于是连续比较，一次遍历中的一个大值将移动几个位置，但是一个小值可能只移动一个位置。在第一次遍历中，最大的值保证能沉到该数组的底端，即保存为  $a[9]$  的值。在第二次遍历中，次最大的值保证能沉到  $a[8]$  上。在第 9 次遍历中，第 9 大的值将沉入  $a[1]$  中。这样就将最小的值保留在  $a[0]$  中，于是只用了 9 次遍历就可完成一个 10 元素数组的排序。

排序由 `sort` 方法中的一个嵌套 `for` 循环完成（第 20 行）。如果需要一次交换，可以执行下面 3 条语句：

```

hold = a[ i ];
a[ i ] = a[ i+1 ];
a[ i+1 ] = hold;

```

其中变量 `hold` 用于临时存放两个交换值中的一个。交换不能只用两条赋值语句来完成：

```

a[ i ] = a[ i+1 ];
a[ i+1 ] = a[ i ];

```

例如，如果 `a[i]` 是 7，`a[i+1]` 是 5，经过第一次赋值后，两个值将都是 5，这样将丢失 7。因此需要一个中间变量 `hold`。

冒泡排序的主要优点是易于编程，但是，冒泡排序的速度较慢，对于大数组排序尤为明显。在练习中，我们将开发效率更高的冒泡排序版本，同时研究比冒泡排序效率更高的排序方法。在一些更高级的课程（例如数据结构）中，将更深入地探讨排序和查找方法。

## 5.8 数组查找：线性查找和二分查找

通常，一个程序员要处理数组中的大量数据。有时，需要确定一个数组是否包含一个能匹配关键值（key value）的数值。找出数组中特定元素的过程称为查找，本节我们讨论两种查找技术——简单的线性查找和更有效的二分查找。本章最后的练习 5.31 和 5.32 要求读者实现线性查找和二分查找的递归版本。

### 5.8.1 线性查找

线性查找（如图 5.14 所示）利用查找关键值与每一个数组元素进行比较。由于数组不是处于某种特定的顺序，因此在第一个元素中发现该值和在最后一个元素中发现该值的可能性是一样的。平均情况下，程序必须同数组的一半元素进行比较，然后才能找到与查找关键值匹配的值。

线性查找法适用于小数组或未排序的数组。但是，对于大数组而言，线性查找法就不够有效了。如果数组是有序的，则可以使用高速的二分查找技术。

---

```

1 // Fig. 5.14: LinearSearch.java
2 // Linear search of an array
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class LinearSearch extends Applet {
7     int a[ ];
8     int element;
9     String searchKey;
10    Label enterLabel;
11    TextField enter;
12    Label resultLabel;
13    TextField result;
14
15    public void init()
16    {
17        a = new int[ 100 ];
18
19        for ( int i = 0; i < a.length; i++ ) // create data

```



```

20         a[ i ] = 2 * i;
21
22         enterLabel = new Label( "Enter integer search key" );
23         enter = new TextField( 10 );
24         resultLabel = new Label( "Result" );
25         result = new TextField( 25 );
26         result.setEditable( false );
27         add( enterLabel );
28         add( enter );
29         add( resultLabel );
30         add( result );
31     }
32
33     public int linearSearch( int key )
34     {
35         for ( int n = 0; n < a.length; n++ )
36             if ( a[ n ] == key )
37                 return n;
38
39         return -1;
40     }
41
42     public boolean action( Event event, Object o )
43     {
44         if ( event.target == enter ) {
45             searchKey = event.arg.toString();
46             element = linearSearch( Integer.parseInt( searchKey ) );
47
48             if ( element != -1 )
49                 result.setText( "Found value in element " + element );
50             else
51                 result.setText( "Value not found" );
52         }
53
54         return true;
55     }
56 }

```

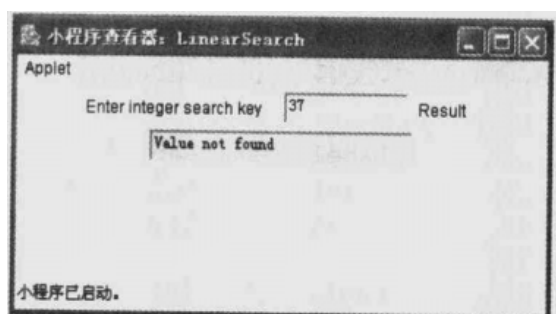
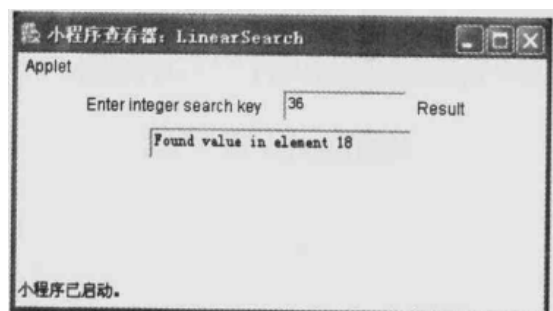


图 5.14 数组的线性查找

## 5.8.2 二分查找

二分查找算法通过比较被查找数组中的一个元素后,就能去除数组中一半元素的可能性。该算法找到中间的数组元素并将其与查找关键值相比较。如果它们相等,则找到查找关键值,并且返回

该元素的下标。否则,问题就变为在数组的一半元素中查找。如果查找关键值小于中间的数组元素,就查找数组的前半部分,否则查找数组的后半部分。如果查找关键值不是要查找的子数组(原数组的一部分)的中间元素,该算法就基于原数组的四分之一元素进行处理。查找将一直继续,直到查找关键值等于某子数组的中间元素,或者该子数组中包含着一个不等于查找关键值的元素(即未找到查找关键值)。

即使是最糟糕的情况下,使用二分查找处理一个具有 1 024 个元素的数组只需比较 10 次。反复用 1 024 除以 2 (因为每次比较后都能去除一半元素),则产生 512、256、128、64、32、16、8、4、2 和 1。数值 1 024 ( $2^{10}$ ) 不断除以 2, 只需 10 次就得到了 1。除以 2 就相当于在二分查找算法中比较一次。在一个有 1 048 576 ( $2^{20}$ ) 个元素的数组中,找到关键值最多只进行 20 次比较。具有 10 亿个元素的数组只用 30 次就能找到关键值。与平均要比较数组的一半元素的线性查找相比,这是一个性能上的飞跃。对于一个包含 10 亿个元素的数组而言,其差别为平均 5 亿次的比较和 30 次的比较。对于二分查找而言,任何有序数组的最大比较次数是  $n$ ,  $n$  是使  $2^n$  大于等于数组中元素个数的最小值。

图 5.15 提供了方法 `binarySearch` (第 64 行) 的迭代版本。该方法接收两个参数: 一个整数 `key` (查找关键值) 以及 `graphics` 对象 `gg` (用于输出)。如果 `key` 没有与子数组的中间元素匹配,则调整 `low` 下标或 `high` 下标 (均在方法中声明), 使查找范围缩减至一个较小的子数组上。如果 `key` 小于中间元素,则将 `high` 下标置为 `middle - 1`, 然后继续在 `low` 到 `middle - 1` 的元素上继续查找。如果 `key` 大于中间元素,则将 `low` 下标置为 `middle + 1`, 接着在 `middle + 1` 到 `high` 的元素上继续进行查找。该程序使用了一个 15 元素的数组。大于数组元素个数的 2 的幂次值为 16 ( $2^4$ ), 因此最多只需 4 次就可找到 `key`。`printRow` 方法在二分查找过程中输出每一个子数组。每一个子数组的中间元素用星号 (\*) 标出, 以表明 `key` 将同它进行比较。

```

1 // Fig. 5.15: BinarySearch.java
2 // Binary search of an array
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class BinarySearch extends Applet {
7     int a [ ];
8     int element;
9     String searchKey;
10    int xPosition; // applet horizontal drawing position
11    int yPosition; // applet vertical drawing position
12    Label enterLabel;
13    TextField enter;
14    Label resultLabel;
15    TextField result;
16    boolean timeToSearch = false;
17
18    public void init()
19    {
20        a = new int [ 15 ];
21
22        for ( int i = 0; i < a.length; i++ ) // create data
23            a [ i ] = 2 * i;
24
25        enterLabel = new Label ( "Enter key" );
26        enter = new TextField ( 5 );
27        resultLabel = new Label ( "Result" );
28        result = new TextField ( 22 );

```

```
29         result.setEditable( false );
30         add( enterLabel );
31         add( enter );
32         add( resultLabel );
33         add( result );
34     }
35
36     public void paint( Graphics g )
37     {
38         if ( timeToSearch ) { // prevents search 1st time called
39             element = binarySearch(
40                 Integer.parseInt( searchKey ), g );
41
42             if ( element != -1 )
43                 result.setText(
44                     "Found value in element " + element );
45             else
46                 result.setText( "Value not found" );
47         }
48     }
49
50     public boolean action( Event event, Object o )
51     {
52         if ( event.target == enter ) {
53             timeToSearch = true;
54             xPosition = 25;
55             yPosition = 75;
56             searchKey = event.arg.toString();
57             repaint(); // call paint to start search and output
58         }
59
60         return true;
61     }
62
63     // Binary search
64     public int binarySearch( int key, Graphics gg )
65     {
66         gg.drawString( "Portions of array searched",
67             xPosition, yPosition );
68         yPosition += 15;
69
70         int low = 0;           // low subscript
71         int high = a.length - 1; // high subscript
72         int middle;           // middle subscript
73
74         while ( low <= high ) {
75             middle = ( low + high ) / 2;
76
77             printRow( low, middle, high, gg );
78
79             if ( key == a[ middle ] ) // match
80                 return middle;
81             else if ( key < a[ middle ] )
82                 high = middle - 1; // search low end of array
83             else
84                 low = middle + 1; // search high end of array
```

```

85         }
86
87         return -1;    // searchKey not found
88     }
89
90     // Print one row of output showing the current
91     // part of the array being processed.
92     void printRow( int low, int mid, int high, Graphics gg )
93     {
94         xPos = 25;
95
96         for ( int i = 0; i < a.length; i++ ) {
97             if ( i < low || i > high )
98                 gg.drawString( "", xPos, yPos );
99             else if ( i == mid )    // mark middle value
100                 gg.drawString( String.valueOf( a[ i ] ) + "*",
101                               xPos, yPos );
102             else
103                 gg.drawString( String.valueOf( a[ i ] ),
104                               xPos, yPos );
105
106             xPos += 20;
107         }
108
109         yPos += 15;
110     }
111 }

```

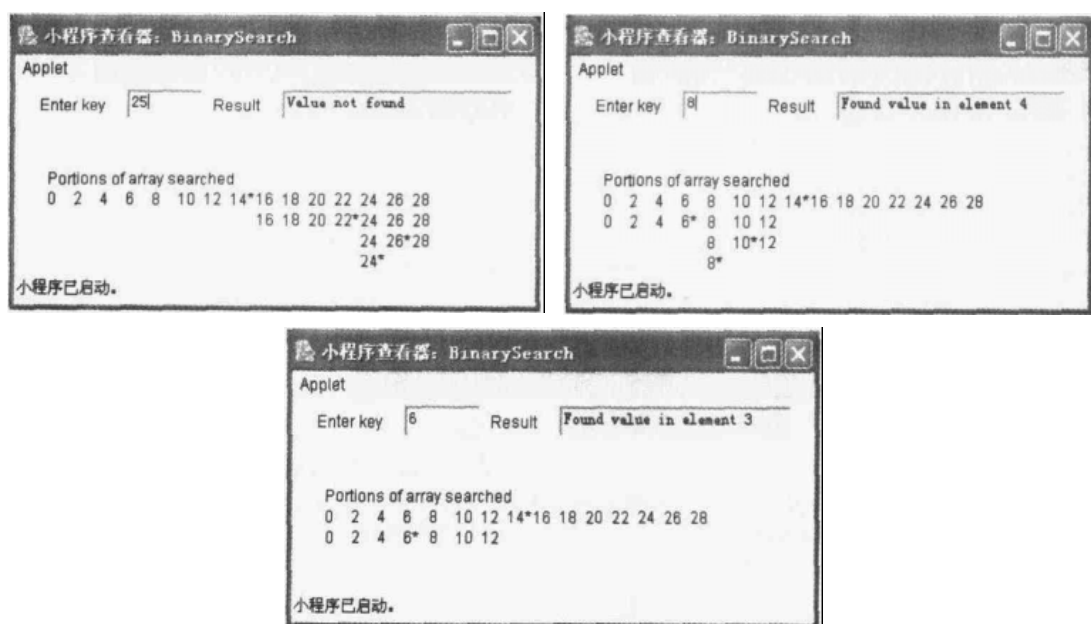


图 5.15 有序数组的二分查找

## 5.9 多维数组

带有两个下标的多维数组常常用于表示按行和列排列的信息值。为了标识一个列表元素,我们必须规定两个下标:第一个标识元素的行(习惯用法),第二个指明元素的列。需要两个下标来标

识某一特定元素的表或数组称为二维数组。注意，多维数组可以拥有两个以上的下标。Java并不直接支持多维数组，但允许程序员定义其元素也是单下标数组的单下标数组，从而实现同样效果。

图 5.16 说明了一个二维数组 *a*，它包括 3 行 4 列，于是称之为 3 乘 4 数组。一般情况下，一个有 *m* 行 *n* 列的数组称为 *m* 乘 *n* 数组。

	0列	1列	2列	3列
0行	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
1行	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
2行	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

列下标  
——行下标  
· 数组名

图 5.16 一个有 3 行 4 列的二维数组

在图 5.16 中，数组 *a* 的每个元素由一个元素名形式 *a*[ *i* ][ *j* ] 标识出来。*a* 是数组名，*i* 和 *j* 是唯一标识 *a* 中每一元素的下标。注意，第一行中数组元素名的第一个下标都为 0，第四列中元素名的第二个下标都为 3。

多维数组可以像一个单下标数组那样初始化。一个二维数组 *b*[ 2 ][ 2 ] 可以利用下列语句进行声明和初始化：

```
int b[ ][ ] = { { 1, 2 }, { 3, 4 } };
```

元素值按行分组并由花括号括起来。因此，*b*[0][0] 和 *b*[0][1] 的初始值为 1 和 2，*b*[1][0] 和 *b*[1][1] 的初始值为 3 和 4。

多维数组可看成是数组的数组。声明：

```
int b[ ][ ] = { { 1, 2 }, { 3, 4, 5 } };
```

创建了一个整数数组 *b*，0 行有两个元素（1 和 2），1 行包括三个元素（3，4 和 5）。

图 5.17 展示了在声明中初始化二维数组的方法，该程序声明了两个数组。

```
1 // Fig. 5.17: InitArray.java
2 // Initializing multidimensional arrays
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class InitArray extends Applet {
7     int array1[ ][ ] = { { 1, 2, 3 }, { 4, 5, 6 } };
8     int array2[ ][ ] = { { 1, 2 }, { 4 } };
9
10    // paint the applet
11    public void paint( Graphics g )
12    {
13        g.drawString( "Values in array1 by row are", 25, 25 );
14        printArray( array1, g, 40 );
15
16        g.drawString( "Values in array2 by row are", 25, 70 );
```

```

17         printArray( array2, g, 85 );
18     }
19
20     public void printArray( int a[ ][ ], Graphics g, int y )
21     {
22         int x = 25;
23
24         for ( int i = 0; i < a.length; i++ ) {
25
26             for ( int j = 0; j < a[ i ].length; j++ ) {
27                 g.drawString( String.valueOf( a[ i ][ j ] ), x, y );
28                 x += 15;
29             }
30
31             x = 25;
32             y += 15;
33         }
34     }
35 }

```

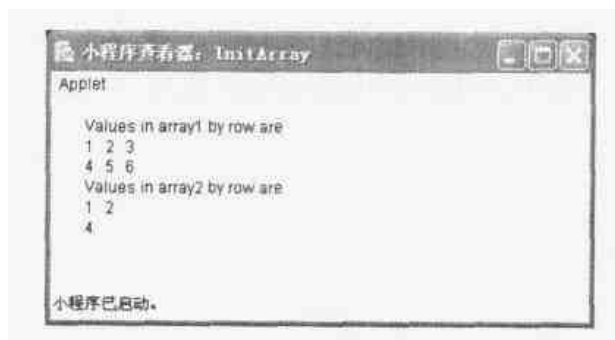


图 5.17 初始化多维数组

array1 的声明中用两个子表提供了 6 个初始化值。第一个子表将数组第一行初始化为 1、2 和 3；第二个子表将第二行初始化为 4、5 和 6。array2 的声明用两个子表提供了 3 个初始化值。第一个子表将第一行的元素显式地初始化为 1 和 2。第二个子表将第二行的值初始化为 4。

该程序调用方法 printArray (第 14 行和第 17 行) 输出每一个数组的元素。注意，在该方法的定义中将数组参数声明为 int a[ ][ ], 指明将接收一个二维数组参数。一个 Graphics 引用也作为参数传递，以使 PrintArray 方法能够将每个数组的内容输出到 applet 上。注意，这里使用了一个嵌套的 for 循环结构输出二维数组的每一行。在外层 for 结构中，表达式 a.length 确定了数组中行的个数。在内层 for 结构中，表达式 a[ i ].length 确定了在该数组中每一行的列数。许多常见的数组操作都使用 for 循环结构，例如，下面的 for 结构将图 5.16 中数组 a 的第三行的所有元素置为 0：

```

for ( int col = 0; col < a[ 2 ].length; col++ )
    a[ 2 ][ col ] = 0;

```

我们所指的是第三行，因此知道第一个下标总是 2 (0 是第一行，1 是第二行)。for 循环只改变第二个下标 (即列下标)。前面的 for 结构等效于下面的赋值语句：

```

a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;

```

下面的嵌套 for 结构确定了数组 a 的所有元素值的总和:

```
total = 0 ;
for ( int row = 0 ; row < a.length ; row++ )
    for ( int col = 0 ; col < a[ row ].length ; col++ )
        total += a[ row ][ col ] ;
```

for 结构每次处理数组的一行元素。外层的 for 结构一开始便将 row 下标置为 0, 于是第一行的元素就可由内层的 for 结构求和。外层 for 结构接着将 row 增长为 1, 于是又可以加上第二行的和。当嵌套的 for 结构结束时, 就可以得到所有元素之和。

图 5.18 中的程序在  $3 \times 4$  数组 grade 上执行了几个其他的常见操作。每行代表一名学生, 每列代表该学生在本学期中所选的 4 门功课的成绩。minimum 方法确定了所有学生在本学期中的最低成绩, maximum 方法确定了所有学生在本学期中的最高成绩, average 方法确定了某一个学生的学期平均成绩, printArray 方法以表格形式输出二维数组。minimum 方法、maximum 方法和 printArray 方法都使用了 grades 数组、students 变量 (数组中的行数) 以及 exams (数组中的列数) 变量, 每个方法都使用 for 结构遍历数组 grades。下面的嵌套 for 结构是方法 minimum 的定义:

```
int lowGrade = 100 ;
for ( int i = 0 ; i < students ; i++ )
    for ( int j = 0 ; j < exams ; j++ )
        if ( grades[ i ][ j ] < lowGrade )
            lowGrade = grades[ i ][ j ] ;
```

外层 for 结构一开始就将 i (即行下标) 置为 0, 于是第一行的元素就在内层 for 结构体中同变量 lowGrade 相比较。内层的 for 结构遍历某一学生的 4 个成绩, 并将每个成绩同 lowGrade 比较, 如果成绩低于 lowGrade, 就将 lowGrade 置为该成绩。外层的 for 结构接着对行下标加 1, 第二行的元素又同变量 lowGrade 相比较。外层的 for 结构接着又使行下标变成 2, 第三行的元素再同变量 lowGrade 相比较。当嵌套结构执行完时, lowGrade 就含有该二维数组中的最低成绩。maximum 方法同 minimum 方法的原理相似。

average 方法带有一个参数, 即针对某个学生的测验结果的单下标数组。当调用 average 时, 这个参数就是描述二维数组 grades 中某一特定行的 grades[i], 它将传递给 average。例如, 参数 grades[1] 代表了 4 个值 (一个单下标的成绩数组), 它们存储在二维数组 grades 的第二行中。在 Java 中, 一个二维数组就是指元素为单下标数组的数组。average 方法计算出数组元素的和, 并用总和除以功课数, 最后返回这个浮点数结果 (double)。

```
1 // Fig. 5.18: DoubleArray.java
2 // Double-subscripted array example
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class DoubleArray extends Applet {
7     int grades[ ][ ] = { { 77, 68, 86, 73 },
8                           { 96, 87, 89, 81 },
9                           { 70, 90, 86, 81 } };
10    int students, exams;
11    int xPosition, yPosition;
12
13    // initialize instance variables
14    public void init()
```

```

15     {
16         students = grades.length;
17         exams = grades[ 0 ].length;
18     }
19
20     // paint the applet
21     public void paint( Graphics g )
22     {
23         xPosition = 25;
24         yPosition = 25;
25
26         g.drawString( "The array is:", xPosition, yPosition );
27         yPosition += 15;
28         printArray( g );
29         xPosition = 25;
30         yPosition += 30;
31         g.drawString( "Lowest grade:", xPosition, yPosition );
32         int min = minimum();
33         g.drawString( String.valueOf( min ),
34                     xPosition + 85, yPosition );
35         yPosition += 15;
36         g.drawString( "Highest grade:", xPosition, yPosition );
37         int max = maximum();
38         g.drawString( String.valueOf( max ),
39                     xPosition + 85, yPosition );
40         yPosition += 15;
41
42         for ( int i = 0; i < students; i++ ) {
43             g.drawString( "Average for student " + i + " is ",
44                         25, yPosition );
45             double ave = average( grades[ i ] );
46             g.drawString( String.valueOf( ave ), 165, yPosition );
47             yPosition += 15;
48         }
49     }
50
51     // find the minimum grade
52     public int minimum()
53     {
54         int lowGrade = 100;
55
56         for ( int i = 0; i < students; i++ )
57             for ( int j = 0; j < exams; j++ )
58                 if ( grades[ i ][ j ] < lowGrade )
59                     lowGrade = grades[ i ][ j ];
60
61         return lowGrade;
62     }
63
64     // find the maximum grade
65     public int maximum()
66     {
67         int highGrade = 0;
68
69         for ( int i = 0; i < students; i++ )
70             for ( int j = 0; j < exams; j++ )
71                 if ( grades[ i ][ j ] > highGrade )
72                     highGrade = grades[ i ][ j ];
73     }

```



```

74         return highGrade;
75     }
76
77     // determine the average grade for a particular
78     // student (or set of grades)
79     public double average( int setOfGrades[ ] )
80     {
81         int total = 0;
82
83         for ( int i = 0; i < setOfGrades.length; i++ )
84             total += setOfGrades[ i ];
85
86         return (double) total / setOfGrades.length;
87     }
88
89     // print the array
90     public void printArray( Graphics g )
91     {
92         xPosition = 80;
93
94         for ( int i = 0; i < exams; i++ ) {
95             g.drawString( "[" + i + "]", xPosition, yPosition );
96             xPosition += 30;
97         }
98
99         for ( int i = 0; i < students; i++ ) {
100             xPosition = 25;
101             yPosition += 15;
102             g.drawString( "grades[" + i + "]",
103                           xPosition, yPosition );
104             xPosition = 80;
105
106             for ( int j = 0; j < exams; j++ ) {
107                 g.drawString( String.valueOf( grades[ i ][ j ] ),
108                               xPosition, yPosition );
109                 xPosition += 30;
110             }
111         }
112     }
113 }

```

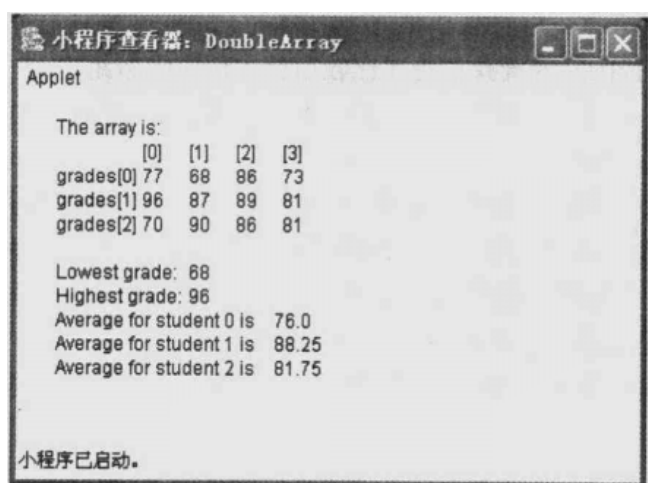


图 5.18 使用双下标数组的例子

## 小结

- Java 用数组存储值列表。一个数组是连续的相关内存空间的集合。这些空间都与这样一个事实有关，即都拥有相同的名字和类型。为了访问一个数组的特定位置空间或元素，我们规定了数组的名字和元素的下标。
- 在创建数组对象时可以通过指定 `length` 成员来设置数组元素数。
- 下标可以是一个整数或一个整数表达式。如果使用一个表达式作为下标，那么要计算该表达式以确定该数组的特定元素。
- Java 数组总是从元素 0 开始，因此要特别注意引用数组的第 7 个元素不同于数组元素 7，第 7 个元素的下标为 6，而数组元素 7 的下标则为 7（实际上是数组的第 8 个元素）。这是产生“差 1 错误”的根源。
- 数组要占用存储空间。为了保存整数数组 `b` 的 100 个元素和整数数组 `x` 的 27 个元素，可以使用下面的语句：

```
int b[] = new int [ 100 ] , x[] = new int [ 27 ]
```

- 数组元素的初始化可以通过声明、赋值和输入来完成。
- Java 能防止越界访问数组。
- 如果要向方法传递数组，则应传递数组的名字。如果要向方法传递数组的单个元素，则需传递后跟特定元素下标（在方括号中）的数组名字。
- 数组通过按引用调用来传递，因此被调用的方法能够修改调用者中原始数组的元素值。单个数组元素通过按值调用来传递。
- 如果要接收一个数组参数，则该方法的参数表必须说明要接收的数组。
- 一个数组可以通过冒泡排序技术进行排序。这需要对数组进行几次遍历。在每一次遍历中，比较一对相邻的元素。如果一对相邻元素是有序的（或值是相同的），则保留原有位置。如果一对相邻元素是无序的，则交换它们的位置。对于小数组而言，冒泡排序是可以接受的；但对于较大的数组而言，相比于更高级的排序算法，冒泡排序就显得效率较低。
- 线性查找比较查找关键值与数组中的每个元素。如果数组没有按某种顺序排序，那么在第一个元素和最后一个元素上找到该值的可能性是一样的。平均而言，此类程序将比较查找关键值与数组的一半元素。线性查找很适合小数组，甚至对大型无序数组也是可以接受的。
- 对于有序数组而言，二分查找在每次比较后就可以去除数组中的一半元素。这种算法定位数组的中间元素，并将其同查找关键值相比较。如果相等，说明已找到查找关键值，并返回这个元素的下标；否则继续查找剩下的一半数组。
- 在最糟的情况下，使用二分查找处理一个有序的 1 024 元素的数组，只需进行 10 次比较。
- 数组可以用来表示以行和列形式组织的信息值表。为了标识表中的一个特定元素，需要规定两个下标：第一个标识所包含元素的行，第二个标识所包含元素的列。需要使用两个下标来标识出一个特定元素的表或者数组称为二维数组。
- 如果要向一个接收单下标数组的方法传递二维数组的一行，只需传递后面跟有行下标的数组名即可。

## 术语

array	数组	off-by-one error	差1错误
array initializer list	数组初始化值列表	pass-by-reference	按引用传递
binary search for an array	一个数组的二分查找	pass-by-value	按值传递
bounds checking	边界检查	pass of a bubble sort	冒泡排序的一次遍历
bubble sort	冒泡排序	passing arrays to methods	向方法传递数组
column subscript	列下标	position number	位置序号
constant variable	常量变量	row subscript	行下标
declare an array	声明一个数组	search key	查找关键值
double-subscripted array	二维数组	searching an array	查找一个数组
element of an array	数组元素	single-subscripted array	单下标数组
final		sinking sort	下沉排序
initialize an array	初始化一个数组	sorting	排序
initializer	初始化值	sorting an array	排序一个数组
linear search of an array	一个数组的线性查找	square bracket[]	方括号[]
lvalue	左值	subscript	下标
m-by-n array	m乘n数组	table of values	值的表
multiple-subscripted array	多维数组	tabular format	表格形式
name of an array	一个数组的名字	temporary area for exchange of values	用于值交换的临时区域
named constant	命名常量	value of an element	元素的值
		zeroth element	第0个元素

## 自测练习

### 5.1 填空：

- 列表和表格形式的值存放在\_\_\_\_\_中。
- 数组的各元素通过某种约定相关联，即元素具有相同的\_\_\_\_\_和\_\_\_\_\_。
- 用于引用数组中某一特定元素的序号称为数组的\_\_\_\_\_。
- 按顺序排列一个数组元素的过程称为数组\_\_\_\_\_。
- 判断数组是否包含一个特定关键值的过程称为数组\_\_\_\_\_。
- 使用两个下标的数组是一个\_\_\_\_\_数组。

### 5.2 判断下列问题是否正确。如果不正确，请解释原因。

- 数组能够存储不同类型的值。
- 数组下标通常是float类型的。
- 如果将单个的数组元素传递给一个方法，并在方法中将其修改，则在被调用方法执行完之后，该元素的值将改为修改后的值。

### 5.3 假设有一个数组 fractions，编写语句完成下列问题：

- 定义一个常量变量 arraySize 并将其初始化为 10。

- b) 声明一个有 `arraySize` 个元素的 `float` 类型数组，并将元素初始化为 0。
  - c) 给出从数组开头计算的第 4 个元素的名字。
  - d) 引用数组元素 4。
  - e) 将值 1.667 赋给数组元素 9。
  - f) 将值 3.333 赋给数组的第 7 个元素。
  - g) 用一个 `for` 循环结构求出所有数组元素的和。定义整数变量 `x` 作为循环的控制变量。
- 5.4 假设有一个数组 `table`，编写语句完成下列问题：
- a) 声明一个有 3 行 3 列的整数数组。假定常量变量 `arraySize` 已定义为 3。
  - b) 该数组包含多少个元素？
  - c) 用一个 `for` 循环结构将每个数组元素初始化为其下标之和。假定整数变量 `x` 和 `y` 声明为控制变量。
- 5.5 找出下面程序段中的错误并改正。
- ```

a) arraySize = 10 ; // arraySize was declared final;
b) 假定 int b[] = new int[ 10 ];

    for ( int i = 0 ; i <= b.length ; i++ )
        b[i] = 1;

c) 假定 int a[][] = {{1, 2}, {3, 3}};
    a[1, 1] = 5 ;
  
```

## 自测练习答案

- 5.1 a) 数组。b) 名字、类型。c) 下标。d) 排序。e) 查找。f) 二维。
- 5.2 a) 不正确。一个数组只能存储相同类型的值。
- b) 不正确。一个数组下标应当是一个整数或整数表达式。
- c) 不正确。数组的单个基本类型元素是按值调用的。如果将整个数组传递给方法，那么对数组元素的任何修改都将影响到原始数组。同样，对于单个的类元素而言，则是通过按引用调用传递的，对于对象的任何改变将影响到原始数组元素。
- 5.3 a) `final int arraySize = 10;`
- b) `float fraction[] = new float[ arraySize ];`
- c) `fractions[ 3 ]`
- d) `fractions[ 4 ]`
- e) `fractions[ 9 ] = 1.667`
- f) `fractions[ 6 ] = 3.333`
- g) `float total = 0;`
- ```

    for ( int x = 0; x < fractions.length; x++ )
        total += fractions[ x ] ;
  
```
- 5.4 a) `int table[ ][ ] = new int[ arraySize ][ arraySize ];`
- b) 9 个。
- c) `for ( int x = 0; x < table.length; x++ )`
- ```

    for (int y=0; y < table[ x ].length; y++ )
        table[ x ][ y ] = x + y;
  
```

- 5.5 a) 错误: 使用赋值语句给常量变量赋值。  
改正: 用 `final int arraySize` 声明为该常量变量赋值。
- b) 错误: 在数组边界外引用一个数组元素 (`b[10]`)。  
改正: 将 `<=` 运算符改为 `<`。
- c) 错误: 数组下标书写得不正确。  
改正: 将此语句改为 `a[1][1] = 5`。

## 练习

### 5.6 填空:

- a) Java 将列表形式的值存储在 \_\_\_\_\_。
- b) 一个数组的元素与某种事实相关, 它们是 \_\_\_\_\_。
- c) 当引用一个数组元素时, 方括号中的位置序号称为 \_\_\_\_\_。
- d) 数组 `p` 的 4 个元素的名字分别为 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- e) 命名一个数组, 给出其类型并确定数组中的元素个数称为数组 \_\_\_\_\_。
- f) 将一个数组的元素排列成升序或降序的过程称为 \_\_\_\_\_。
- g) 在一个二维数组中, 第一个下标标识一个元素的 \_\_\_\_\_, 第二个下标标识一个元素的 \_\_\_\_\_。
- h) 一个  $m$  乘  $n$  数组包含 \_\_\_\_\_ 行、\_\_\_\_\_ 列、\_\_\_\_\_ 个元素。
- i) 数组 `d` 的行 3、列 5 的元素名是 \_\_\_\_\_。

### 5.7 判断下列问题是否正确。如果不正确, 请解释原因。

- a) 在一个数组中引用特定的位置或元素, 需要给出该数组的名字和该元素的值。
- b) 一个数组声明为数组保留了空间。
- c) 为了说明要为数组 `p` 保留 100 个位置, 程序员编写了以下声明:  
`p[ 100 ] ;`
- d) 在 Java 程序中, 要将一个包含 15 个元素的数组中所有的元素初始化为 0, 则至少要包含一条 `for` 语句。
- e) 在 Java 程序中要计算二维数组元素之和, 则必须包括嵌套的 `for` 语句。

### 5.8 写出完成下列任务的 Java 语句:

- a) 显示字符数组 `f` 第 7 个元素的值。
- b) 将 5 元素单下标数组 `g` 中的每个元素初始化为 8。
- c) 计算浮点数数组 `c` 的 100 个元素之和。
- d) 将 11 元素数组 `a` 复制到含有 34 个元素的数组 `b` 的前面。
- e) 确定并打印含有 99 个元素的浮点数数组 `w` 的最大值和最小值。

### 5.9 考虑一个 $2 \times 3$ 的数组 `t`:

- a) 为 `t` 编写一个声明。
- b) `t` 有多少行?
- c) `t` 有多少列?
- d) `t` 有多少个元素?
- e) 写出 `t` 中第二行的所有元素。
- f) 写出 `t` 中第三列的所有元素。

- g) 写出一条语句, 将 `t` 中第 1 行、第 2 列的元素置为 0
  - h) 写出一系列语句, 将 `t` 的每个元素初始化为 0。不要使用循环结构。
  - i) 写出一个嵌套 `for` 结构, 将 `t` 的每个元素初始化为 0
  - j) 写出一条语句, 从终端输入 `t` 的值
  - k) 写出一系列语句, 确定并打印数组 `t` 的最小值
  - l) 写出一条语句, 显示 `t` 的第 1 行元素
  - m) 写出一条语句, 统计 `t` 的第 4 列元素的和
  - n) 写出一系列语句, 使用清晰的表格形式打印出数组 `t`。将列下标作为列标题, 将行下标放在每行的左边
- 5.10 使用一个单下标数组解决下面的问题。一个公司以佣金为基础支付其销售人员的薪金: 销售人员每周收入 \$200, 另加本周销售额的 9%。例如, 一个销售人员每周的销售金额如果为 \$5 000, 则收入为 \$200 另加 \$5 000 的 9%, 也就是 \$650。编写一个程序 (使用一个计数器数组), 确定有多少销售人员的薪金在如下范围 (假定每名销售人员的薪金都将取整):
- a) \$200 ~ \$299
  - b) \$300 ~ \$399
  - c) \$400 ~ \$499
  - d) \$500 ~ \$599
  - e) \$600 ~ \$699
  - f) \$700 ~ \$799
  - g) \$800 ~ \$899
  - h) \$900 ~ \$999
  - i) \$1 000 及更高
- 5.11 图 5.13 中提供的冒泡排序对于大数组的运行效率太低, 可以进行以下简单的改动, 从而增加冒泡排序算法的性能。
- a) 在第一次排序后, 最大的数保证处于数组的最高序号上; 在第二次排序后, 两个最高序号都已经确定了其中的元素, 依次类推。不必每次都进行九次比较, 修改冒泡排序以保证在第二次遍历中只进行八次比较, 第三次遍历中进行七次比较, 依次类推。
  - b) 数组中的数据可能已排好序或近似排好序了, 如果可能使比较的次数更少, 为什么还要比较九次呢? 修改排序算法, 以检查每次遍历结束后是否交换过元素。如果没有, 那么数据必定已经排好序了, 于是程序应当终止。如果交换过元素, 那么至少还要遍历一次。
- 5.12 编写语句, 完成下列单下标数组的操作:
- a) 将整数数组 `counts` 的 10 个元素设置为 0。
  - b) 将整数数组 `bonus` 的 15 个元素分别加 1。
  - c) 用列格式打印整数数组 `bestScores` 的 5 个值。
- 5.13 使用一个单下标数组来解决下列问题。读入 20 个数, 每一个都在 10 和 100 之间 (含 10 和 100)。当读入一个数时, 仅当它不同于已读入的数时才将其打印出来。假定“最糟”的情况是 20 个数全都不同, 请使用尽可能小的数组来解决该问题。
- 5.14 标出  $3 \times 5$  二维数组 `sales` 的元素, 指出在下列程序段中将其置成为 0 的顺序:

```

for ( int row = 0; row < sales.length; row++ )
    for ( int col = 0; col < sales[ row ].length; col++ )
        sales[ row ][ col ] = 0;

```

- 5.15 编写一个程序，模拟掷两个骰子。程序可使用 `Math.random` 来掷第一个骰子，再使用 `Math.random` 掷第二个骰子，应当算出两值之和。注意：由于每个骰子可以显示从 1 到 6 的整数值，因此两值之和将从 2 到 12 变化，其中 7 是最常见的，2 和 12 不易出现。图 5.19 给出了两个骰子和的 36 种可能组合。程序中应当掷骰子 36 000 次，用一个单下标数组计算每种可能组合出现的次数，并用表格形式打印出结果。另外，判断结果是否合理，例如，有 6 种方式滚动到 7，所以大约有六分之一的滚动结果是 7。

|   | 1 | 2 | 3 | 4  | 5  | 6  |
|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

图 5.19 滚动两个骰子的 36 种可能结果

- 5.16 下面这段程序实现了什么操作？

```

1 //Exercise 5.16
2 //What does this program do?
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class MysteryClass extends Applet {
7     int result;
8
9     public void start()
10    {
11        int a[] = { 1,2,3,4,5,6,7,8,9,10 };
12        result=wha Is This(a,a.length);
13
14    }
15
16    public void paint( Graphics g )
17    {
18        g.drawString( "Result is " + result, 25, 25, );
19    }
20
21    public int wahtIsThis( int b[], int size )
22    {
23        if( size == 1)
24            return b[ 0];
25        else
26            return b[ size-1] + whatIsThis( b, size-1 );
27    }
28    ;

```

- 5.17 编写一个程序，玩 1 000 次掷骰子游戏，并回答下列问题：

- a) 有多少次是在第 1 次、第 2 次、……、第 12 次以及第 12 次之后赢的?
- b) 有多少次是在第 1 次、第 2 次、……、第 12 次以及第 12 次之后输的?
- c) 游戏中赢的次数有多少?
- d) 一次游戏的平均长度是多少?
- e) 随着游戏长度的增加, 赢的机会是否增加?

5.18 (航空订票系统) 一家小型航空公司刚刚购买了一台计算机, 用于其最新的自动订票系统。要求读者编写新系统, 为该公司惟一一架飞机 (运量: 10 座) 的每次飞行安排座位, 程序中应当显示下列选项:

```
Please type 1 for "smoking" (吸烟舱) 图标 1
Please type 2 for "nonsmoking" (非吸烟舱) 图标 2
```

如果某人按下 1, 那么程序应当在吸烟舱 (1 座~5 座) 为其分配一个座位。如果某人按下 2, 那么程序应当在非吸烟舱为其分配一个座位 (6 座~10 座)。在程序中应当打印出一张登机卡, 以表明此人的座位号以及它在飞机的吸烟舱还是非吸烟舱。用一个单下标数组描述飞机的订票情况, 将所有数组元素初始化为 0, 表明所有座位是空的。在分配一个座位之后, 设置数组的相应元素为 1, 表明该座位不能再分配。程序中当然不应分配已分配过的座位。当吸烟舱客满后, 程序应当询问此人是否接受安排安排在非吸烟舱; 反之亦然。如果回答肯定, 那么应进行适当的座位安排。如果回答否定, 那么打印消息 "Next flight leaves in 3 hours." (下次航班 3 小时后起飞)

5.19 下面的程序实现了什么操作?

```
1 //Exercise 5.19
2 //What does this program do?
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class MysteryClass extends Applet {
7     int yPosition;
8     int a[] = { 1,2,3,4,5,6,7,8,9,10 };
9
10    public void paint( Graphics g )
11    {
12        yPosition = 25;
13        someMethod( a, 0, g );
14    }
15
16    public void someMethod( int b[], int x, Graphics g )
17    {
18        if( x < b.length ) {
19            someMethod( b, x+1, g );
20            g.drawString( String.valueOf( b[x] ),
21                        25, yPosition );
22            yPosition += 15;
23        }
24    }
25 }
```

5.20 使用一个二维数组解决下面的问题。一个公司有 4 名销售人员 (1 到 4), 分别销售 5 种不同的产品 (1 到 5)。针对所销售的不同种类的产品, 每个销售人员每天递交一份报告。



每份报告包括:

1. 销售人员代码
2. 产品号码
3. 当日该种产品的全部销售额

因此, 每个销售人员每日递交0到5份报告。假定上个月的所有报告的信息都可以得到。编写一个程序, 读取上个月的销售信息, 对人员和产品分别做出统计。所有的总和应存储在一个二维数组 sales 中。处理完上个月的所有信息之后, 用表格形式打印出结果, 其中每行代表某一销售人员, 每列代表某一产品。统计每一行, 以获得上个月的每名销售人员的总销售额。在输出表格中应当包括放在每行右侧和每列底部的总计金额。

- 5.21 ( 龟图 ) Logo 语言是年轻的计算机用户所熟悉的一种语言, 龟图是这种语言的一个著名应用。想像一只机器海龟在 Java 程序的控制之下在房间内移动。海龟有一支画笔, 可以处在两种位置, 笔头朝上或朝下。当画笔朝下时, 海龟画出其运动的轨迹; 当画笔朝上时, 海龟自由移动, 不会写下任何内容。请模拟海龟的操作并创建一个计算机化的图板。使用一个  $20 \times 20$  的数组 floor, 并将其初始化为 0。从一个包含命令的数组中读取命令, 始终跟踪海龟的当前位置以及画笔的朝向。假定海龟总是从位置(0.0)开始移动, 并且画笔朝上。程序中必须处理的海龟命令如下:

| 命令   | 含义                    |
|------|-----------------------|
| 1    | 画笔朝上                  |
| 2    | 画笔朝下                  |
| 3    | 右转                    |
| 4    | 左转                    |
| 5,10 | 向前走 10 步 (或不为 10 的步数) |
| 6    | 打印 $20 \times 20$ 的数组 |
| 9    | 数据结束 (循环结束标志)         |

假如海龟在图板中心附近的某个位置上, 下面的“程序”将在画笔朝上时画出一个  $12 \times 12$  的正方形。

```

2
5,12
3
5,12
3
5,12
3
5,12
5,12
1
6
9

```

当海龟在运动时画笔朝下, 将数组 floor 的相应元素置为 1。当给出 6 号命令 (打印) 时, 若某数组元素为 1, 则显示一个星号, 或者显示其他字符; 若某数组元素为 0, 就显示一个空格。编写一个程序, 实现龟图功能。编写几个龟图程序, 画出有趣的图形, 并增加其他命令, 使龟图语言的功能更强大。

5.22 (骑士旅行)国际象棋中最有趣的难题之一是骑士旅行问题,最早由数学家欧拉(Euler)提出。这个问题是:骑士这枚棋子在空棋盘上到处移动,能否经过64个方格且在每个方格上只经过一次?我们在这里深入讨论一下这个有趣的问题。骑士在棋盘上做L形运动(在一个方向上走过两格然后在垂直的方向上走一格)因此,从一个空棋盘的中间开始,骑士可如图5.20所示完成8种不同的移动:记为0到7)

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 |   |   |   |   |   |   |   |   |
| 1 |   |   |   | 2 |   | 1 |   |   |
| 2 |   |   | 3 |   |   |   | 0 |   |
| 3 |   |   |   |   | K |   |   |   |
| 4 |   |   | 4 |   |   |   | 7 |   |
| 5 |   |   |   | 5 |   | 6 |   |   |
| 6 |   |   |   |   |   |   |   |   |
| 7 |   |   |   |   |   |   |   |   |

图 5.20 骑士的8种移动

- 在纸上画出一个  $8 \times 8$  的棋盘,尝试手工进行骑士的旅行。在要移入的第一个方格中放入1,第二个方格中放入2,第三个放入3,依次类推。在旅行开始前,估计一下能走到多远,记住,一个完整的旅行包含64次移动,最终走了多远?符合以前的估计吗?
- 现在让我们开发一个程序,使骑士在棋盘上旅行。使用一个  $8 \times 8$  二维数组 `board` 表示棋盘,每一个方格初始化为0。我们根据水平和垂直方向描述8种可能的移动。例如,图5.20中的位移0包括水平向右移动两格及垂直向上移动一格。位移2包括向左移动一格再向上移动两格。水平向左和垂直向上用负数表示。8种移动可由两个单下标数组 `horizontal` 和 `vertical` 描述,如下所示:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

设变量 `currentRow` 和 `currentColumn` 表示骑士当前位置的行和列。为了进行 `moveNumber` 型的移动(其中 `moveNumber` 介于0到7之间),程序中应使用语句:

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

使一个计数器从1到64变化,记录下骑士所移到的方块的最新数据。试试每种可能的移动,看看骑士是否已经访问过那个方块,并确保骑士尚未走出棋盘。编写一个程序,使骑士在棋盘上旅行,运行一下该程序,骑士走过了多少步?

- c) 在尝试过编写和运行骑士旅程序之后,读者可能已经获得了一些有价值的想法,我们将使用这些观点来开发一个试探法(或策略),用于移动骑士。试探法并不能保证成功,但一个精细开发的试探法可以大大增加成功的概率。读者可能已经注意到,外层的方格比起靠近棋盘中心的方格,在移动方面更麻烦一些。事实上,最难到达的方格是四角的方格。

凭借直觉,应首先尝试将骑士移到最不容易到达的方格上,从而为最容易到达的方格留下空位。这样,当棋盘的四角都旅行过后,就可以有较大的成功概率来完成骑士旅行。

我们可以根据每个方格的可到达程度来开发一个“可访问试探法”,然后总是将骑士移到(当然是骑士的L形走法)最不容易到达的方格上。我们利用一些数字来定义一个二维数组accessibility,这些数字表明一个格子能访问的格数。在一个空棋盘上,每个中心方格的可访问格数为8,四角方格的可访问格数为2,其他方格具有的可访问格数为3、4或6,如下所示:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

编写一个使用可访问试探法的骑士旅程序,骑士应当总是移向标有最小可访问性的方格。如果面临多种选择,那么该棋子可以移向任何可选的方格。因此,旅行可以从四个角中的任何一个开始(说明:当骑士在棋盘上旅行时,程序应当随着方格不断被占而减小可访问数值。这样,在旅行过程中的任何时候,每一个方格的可访问数值都要精确地保持为此方格能访问的格数)。运行程序的这一版本,能否实现一个完整的旅行?修改此程序并运行64次旅行,每次从棋盘的不同位置开始。总共能获得多少次完整的旅行?

- d) 编写骑士旅行的另一个版本,当遇到难以在两个或多个方格之间做出选择的情况时,通过向前查看那些从连接的方格可达到的方格来决定怎样移动。程序应当移到那些在下次移动时,到达的方格具有最小可访问数值的方格。

5.23 (骑士旅行:强制算法) 在练习5.22中,我们开发了一个骑士旅行问题的解决方案。这个方法使用了所谓的“可访问试探法”生成许多方案,从而提高了执行效率。由于计算机的性能不断增强,因此我们可以直接利用计算机的功能和相对低级的算法来解决更多

的问题。我们称这种方法为强制算法。

- a) 使用随机数产生器,使得骑士在棋盘上任意移动(当然是走L形)。程序应当运行一次,并将最终的棋盘打印出来,这个骑士走了多远?
  - b) 前面的程序很有可能产生了一个相对较短的旅行。现在修改该程序,尝试1 000次旅行,利用一个单下标数组来跟踪每次旅行所走的步数,当程序完成了1 000次旅行尝试后,应当用清晰的表格形式打印出这一信息。最佳的结果是什么?
  - c) 前面的程序很有可能产生了一些“几乎成功”的但不是完整的旅行。现在去除次数限制,简单地让程序运行,直至产生了一个完整的旅行(说明:此版本的程序可能会在功能强大的机器上运行数小时)以表格形式保存每次旅行所走的步数,当找到第一个完整的旅行时,打印出此表格。在产生一个完整的旅行之前,该程序进行了多少次尝试?使用了多少时间?
  - d) 把骑士旅行问题的强制算法同可访问试探法进行比较。哪种方法需要对问题进行更多的研究?哪种算法更难于开发?哪种算法需要更强大的计算机?我们是否能使用试探法(提前)确定得到一个完整的旅行?我们是否能使用强制算法(提前)确定得到一个完整的旅行?从总体上讨论强制算法的优缺点。
- 5.24 (八皇后) 国际象棋的另一个难题是八皇后问题,可以简单地表述为:是否有可能将八个皇后放在空棋盘上,任何一个皇后都不受其他皇后的“攻击”。也就是说,没有两个皇后在同一行、同一列或者同一对角线上。使用练习5.22的思路提出一个解决八皇后问题的试探法。运行你的程序(提示:可以给棋盘上的每个方格设定一值,以表明如果将一个皇后放在这里,会“删除”空棋盘上的多少个方格。每个角上的方格应赋值为22,如图5.21所示)。一旦这些“删除数字”布满64个方格,那么一个合适的试探法应该是:将下一个皇后放在具有最小“删除数字”的方格上。为什么该策略是可行的?

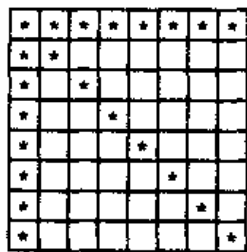


图 5.21 将一皇后放在左上角,有 22 个方格将被删除

- 5.25 (八皇后:强制算法) 在这个练习中,要开发几种强制算法,以解决练习5.24中介绍的八皇后问题。
- a) 解决八皇后问题,使用练习5.23中开发的随机强制算法。
  - b) 使用穷举法,即试出空棋盘上八个皇后的所有可能的组合。
  - c) 为什么穷举法对于解决八皇后问题并不合适?
  - d) 比较和对比随机强制算法和穷举法。
- 5.26 (骑士旅行:封闭旅行试验) 在骑士的旅行中,一个完整的旅行就是骑士移动64次,使其访问到棋盘的每个方格一次且只有一次。一个封闭旅行就是第64次移动将移回骑士开始旅行的方格。修改练习5.22中的程序,如果发生一个完整的旅行,判断其是否是一个封闭旅行。
- 5.27 (女神的筛子) 一个素数是一个只能被自己和1整除的整数。女神的筛子是一个找出素数的方法,操作的过程如下所示:

- 1) 创建一个数组, 将全部元素初始化为1 (true)。含素数下标的数组元素将保持为1, 所有其他元素将最终设置为0 (false)。
  - 2) 从数组下标2 (下标1一定是素数) 开始循环, 每次找到值为1的元素, 在其余的元素中寻找那些下标是该元素下标的整数倍且值为1的元素, 并将其置为零。对于数组下标2而言, 所有数组中在2之后的且下标是2的倍数的元素 (下标4、6、8、10等) 全置为零; 对于数组下标3, 所有在3之后的且下标为3的倍数的元素 (下标6、9、12、15等) 将置为零; 依次类推。当这一过程结束时, 如果数组元素的值仍为1, 则表明该下标是一个素数。这些素数可以打印出来。编写一个程序, 使用含有1000个元素的数组, 确定并打印1到999间的素数。忽略数组中的元素0。
- 5.28 (桶排序) 桶排序是指对一个单下标的正整数数组的排序, 利用一个二维整数数组, 其行下标从0到9, 其列下标从0到 $n-1$ , 其中 $n$ 是要排序的值的个数。二维数组的每一行都称为一个桶。编写一个方法 bucketSort, 将一个整数数组作为参数, 并按照以下步骤执行:
- 1) 根据单下标数组中每个值的个位数字, 将其放入桶数组的某一行中。例如, 97放入行7, 3放入行3, 100放入行0, 我们称其为“分布传递”。
  - 2) 逐行遍历桶数组, 将各行的值复制回原数组, 我们称其为“收集传递”。原值在单下标数组中将按100、3和97的新顺序排列。
  - 3) 重复这一过程 (将个位换成十位、百位、千位等)。在第二次操作中, 将100置于行0, 将3置于行0 (3没有十位数字), 将97置于行9。在收集传递后, 单下标数组中的值的顺序为100、3和97。在第三次操作中, 将100置于行1, 将3置于行0, 将97置于行0 (在3的后面)。在最终的收集传递后, 原始数组就已排好序。

注意, 二维数组的元素数是要排序的整数数组的10倍。实践已证明, 这种排序技术比冒泡排序的性能更好, 但需要更多的内存。冒泡排序只需一个额外的数据元素空间。这就是一个空间与时间交换的例子: 桶排序比冒泡排序需要更多的内存, 但性能更佳。这一版本的桶排序需要在每次扫描后将数据复制回原数组。另一个方法是创建第二个二维数组, 反复在两个桶数组之间交换数据。

## 递归练习

- 5.29 (选择排序) 选择排序的排序过程就是先搜索一个数组, 找出数组中的最小元素, 接着将最小元素同数组中的第一个元素交换。在从第二个元素开始的子数组上重复进行这一过程, 每次搜索都将一个元素置于合适的位置上。这种排序与冒泡排序的性能是相当的, 对于一个 $n$ 元素的数组而言, 要进行 $n-1$ 次比较才能找到最小的值。当要处理的子数组只剩下一个元素时, 数组就将排好序了。编写一个递归方法 selectionSort 来实现该算法。
- 5.30 (回文) 一条回文是一个字符串, 从前向后和从后向前拼读该字符串的结果是一样的。一些回文的例子有: “radar”、“able was i ere i saw elba”以及 “a man a plan a canal panama” (如果忽略空格)。编写一个递归方法 testPalindrome, 如果存放在数组中的字符串是一条回文, 那么就返回1, 否则返回0。该方法应忽略空格和标点。
- 5.31 (线性查找) 修改图 5.14 中的程序, 使用递归方法 linearSearch 来完成数组的线性查找。该方法应当接收一个整数数组以及数组大小作为参数。如果找到了查找键, 就返回数组下标, 否则返回-1。

- 5.32 (二分法查找) 修改图 5.15 中的程序, 使用一个递归方法 `binarySearch` 来完成数组的二分查找。该方法应当接收一个整数数组以及起始和终止下标作为参数。如果找到了查找键, 就返回数组下标, 否则返回 -1。
- 5.33 (八皇后) 修改练习 5.24 中的八皇后程序, 利用递归方法来解决这一问题。
- 5.34 (打印数组) 编写一个递归方法 `printArray`, 使用数组和数组的大小作为参数, 并且没有返回值。该方法在收到一个大小为 0 的数组时停止处理。
- 5.35 (逆向打印字符串) 编写一个递归方法 `stringReverse`, 将一个字符数组作为参数, 逆向打印该字符串, 且该方法没有返回值。当遇到终止空字符 (`null`) 时停止处理。
- 5.36 (找出数组中的最小值) 编写一个递归方法 `recursiveMinimum`, 将一个整数数组和数组大小作为参数, 返回数组的最小值。该方法在接收到一个 1 元素数组时应停止处理。
- 5.37 (快速排序) 在本章的练习和例子中, 我们讨论了诸如冒泡排序、桶排序以及选择排序这样的一些技术。我们现在提供的递归排序技术称为快速排序。对于一个单下标数组值进行排序的基本算法如下:

- 1) 分离步骤: 取出未排序数组的第一个元素, 并确定其在已排序数组中的最终位置, 即所有在该元素左边的数组元素都小于等于该元素, 所有在该元素右边的数组元素都大于该元素。我们现在便有一个处于合适位置的元素, 并加上两个未排序的子数组。
- 2) 递归步骤: 在每一个未排序的子数组上执行步骤 1。每次在一个子数组上执行步骤 1 时, 就会将一个元素放在已排序数组的最终位置上, 同时又有两个未排序的子数组生成。当一个子数组只包含一个元素时, 它必定已排好序, 因此这个元素就在它的最终位置上。基本算法似乎很简单, 但是我们如何判断子数组第一个元素的最终位置呢? 作为一个例子, 考虑下面的一组值 (粗体显示的元素是分离元素, 它将放在已排序数组的最终位置上):

37   2   6   4   89   8   10   12   68   45

- 1) 从数组的最右端开始, 将每个元素同 37 进行比较, 直到发现一个小于等于 37 的元素, 接着把 37 和该元素进行交换。第一个小于 37 的元素是 12, 于是 37 和 12 交换。新数组为:

12   2   6   4   89   10   **37**   68   45

元素 12 用斜体标出, 表明它刚同 37 交换。

- 2) 从数组的左边开始 (但要在 12 之后), 将每个元素同 37 进行比较, 直到找到大于 37 的元素, 然后交换这两个元素。第一个大于 37 的元素是 89, 于是 37 和 89 交换。新的数组为:

12   2   6   4   **37**   8   10   89   68   45

- 3) 从右边开始 (但要在 89 之前), 将每个元素同 37 进行比较, 直到有元素小于 37, 然后交换这两元素。第一个小于 37 的元素是 10, 于是 37 和 10 交换。新的数组为:

12   2   6   4   10   8   **37**   89   68   45

- 4) 从左边开始 (但要在 10 之后), 将每个元素同 37 相比, 直到发现大于 37 的元素, 然后把 37 和那个元素进行交换。这时没有元素大于 37 了, 于是我们就知道 37 已在排序数组的最终位置上。一旦分离完上面的数组, 就将产生两个未排序的子数组。包

含小于37的值的子数组为12、2、6、4、10和8。包含大于37的值的子数组为89、68和45。在分开的两个子数组上以同样方法继续进行排序。

基于前面的讨论,编写一个递归的方法 quickSort, 对一个整数数组进行排序。该方法应当接收整数数组的起始下标和终止下标作为参数。quickSort方法应调用partition方法来完成分离步骤。

5.38 (迷宫旅行) 下面的“#”号和“.”号是一个二维数组,表示一个迷宫。

```

# # # # # # # # # # # #
# . . . # . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . # . . . #
# # # # # # # # # # # #

```

在上面的二维数组中,#号代表迷宫的墙,句点代表走迷宫的可能路径。只能移动到数组中有句点的地方。

有一个简单的走迷宫的算法,并且能保证找到出口(假定有出口)。如果没有出口,就会再次回到出发点。将你的右手放在右边的墙上,开始向前走。不要将你的手从墙上移开。如果迷宫向右转,就跟着向右转,只要你的手不从墙上移开,最终将会到达迷宫的出口。也许有一条路径比你走过的要短,但只要遵循这个算法,保证能走出迷宫。

编写一个递归方法 mazeTraverse 来走出迷宫。该方法的参数应为一个  $12 \times 12$  字符数组,代表迷宫以及迷宫的起始位置。在 mazeTraverse 试图从迷宫找到出口的过程中,应当将字符 x 放入走过路径中的每个方格上。该方法应在每次移动后显示整个迷宫,以便用户能够观察到这个迷宫是如何解开的。

5.39 (随机生成迷宫) 编写一个方法 mazeGenerator, 其参数为一个二维的  $12 \times 12$  字符数组,并随机生成一个迷宫。该方法应当提供迷宫的起止位置。利用随机生成的迷宫试试练习 5.38 中的方法 mazeTraverse。

5.40 (任意大小的迷宫) 将练习 5.38 和练习 5.39 中的方法 mazeTraverse 和 mazeGenerator 通用化,以处理具有任意长度和宽度的迷宫。

5.41 (模拟程序:龟兔赛跑) 在这个问题中,我们将再现经典的龟兔赛跑。程序中使用随机数生成法来开发一个模拟这一著名事件的应用程序。

比赛场地为70个方格,参赛者从“方格1”开始出发。每个方格代表比赛过程中所经过的一个位置。终点为“方格70”。最先到达或通过“方格70”的参赛者将赢得一桶新鲜的胡萝卜和莴苣。在比赛过程中可能会经过一段很滑的山路,所以参赛者可能会滑倒。

程序中有一只时钟,每秒滴答一次。随着每次时钟滴答,程序应该根据下列规则来调整动物的位置:

| 动物 | 移动类型 | 时间百分比 | 实际移动    |
|----|------|-------|---------|
| 龟  | 快速走  | 50%   | 同左 3 格  |
|    | 滑倒   | 20%   | 同左 6 格  |
|    | 慢速走  | 30%   | 同右 1 格  |
| 兔  | 睡觉   | 20%   | 原地不动    |
|    | 大跳   | 20%   | 同右 9 格  |
|    | 大滑倒  | 10%   | 同左 12 格 |
|    | 小跳   | 30%   | 同右 1 格  |
|    | 小滑倒  | 20%   | 同左 2 格  |

使用几个变量来跟踪动物的位置（即位置号 1~70）。在位置 1（即起跑线）上启动每个动物。如果动物在方格 1 前向左滑动，则将动物移回方格 1。

通过产生一个随机整数  $i$  来生成上表中的百分比， $i$  的范围是  $1 \leq i \leq 10$ 。对于乌龟而言，当  $1 \leq i \leq 5$  时“快速走”，当  $6 \leq i \leq 7$  时“打滑”，当  $8 \leq i \leq 10$  时“慢速走”。使用类似的方法来移动兔子。比赛开始时打印以下的字符串：

```
BANG      ( 砰 )
AND THEY'RE OFF ( 它们出发了 )
```

程序继续执行，时钟每滴答一次（即每循环一次），就打印 70 号方格位置的一条线，其中乌龟的位置用 T 表示，兔子的位置用 H 表示。偶尔，竞赛者们会挤到同一个格子 L。此时，乌龟会咬兔子，程序要在这个位置上打印“OUCH!!!”。所有不是 T、H 或 OUCH!!!（偶局情形）的地方都用空格代替。

打印出每行之后，确定每个动物是否到达或穿过了 70 号方格。如果有，则打印出胜者并终止模拟程序。如果是乌龟胜利了，则打印“TORTOISE WINS!!! YAY!!!”，如果是兔子获胜了，则打印“Hare wins, Yuch”。如果两个动物在同一时刻打成平手，那么应当表扬乌龟（因其处于劣势），或者打印“It's a tie”。如果没有动物获胜，就再执行一遍循环来模拟时钟的下一个时刻。

## 特殊小节：建立自己的计算机

在下面的几个练习中，我们将暂时从高级语言的编程世界中转移出来，这里将打开一台计算的外壳，看看它的内部结构。现在，我们介绍一下机器语言编程并编写几个机器语言的程序。为了实现这个具有特殊价值的实验，我们在其后要建立一台计算机（通过基于软件的模拟技术），在其上可以执行 Simpletron 机器语言程序！

**5.42（机器语言编程）** 让我们创建一台称为 Simpletron 的计算机。顾名思义，这是一台简单的机器；但是，正如我们将要看到的，Simpletron 同样也是功能强大的。Simpletron 运行那些只有自身才能理解的语言程序，这种语言就是 Simpletron 机器语言（SML）。

Simpletron 包括一个累加器，即一个特殊的寄存器，其中预先存放了 Simpletron 在计算中要用到的或者要以各种方式对其进行判断的信息。Simpletron 中的信息都是以字的形式处理的。一个字是一个有符号的 4 位十进制数，例如 +3364、-1293、+0007、-0001 等。Simpletron 含有一个 100 字的存储器，可以通过它们的位置号 00、01、…、99 来引用这些字。

在运行一个 SML 程序之前，必须将程序加载或放置到存储器中。每个 SML 程序的第



一条指令（或语句）总是放在单元 00 上，模拟程序将在该单元开始执行。

每一条使用 SML 语言编写的指令都占用 Simpletron 存储器的一个字（因此指令正好是有符号的 4 位十进制数）。我们假定 SML 指令的符号总是正号，但是数据字的符号可为正也可为负。Simpletron 存储器的每个单元可以存放一条指令、一个程序所使用的数据值，或者是一个未使用（即未定义）的存储区。每条 SML 指令的前两位是指定要完成操作的操作码。SML 的操作码均列在图 5.22 中。

| 操作码                               | 含义                              |
|-----------------------------------|---------------------------------|
| 输入/输出操作：                          |                                 |
| static final int READ = 10;       | 从终端读取一个字到存储器的指定单元               |
| static final int WRITE = 11;      | 从存储器的指定单元将一个字写到终端               |
| 加载/存储操作：                          |                                 |
| static final int LOAD = 20;       | 从存储器的指定单元把一个字载入累加器              |
| static final int STORE = 21;      | 把累加器的一个字存放到存储器                  |
| 算术操作：                             |                                 |
| static final int ADD = 30;        | 将存储器指定单元的字加到累加器中（结果存放在累加器）      |
| static final int SUBTRACT = 31;   | 从累加器中减去存储器中指定单元的字（结果存放在累加器中）    |
| static final int DIVIDE = 32;     | 使用累加器中的字除以存储器中指定单元的字（结果存放在累加器中） |
| static final int MULTIPLY = 33;   | 使用累加器中的字乘以存储器中指定单元的字（结果存放在累加器中） |
| 控制转移操作：                           |                                 |
| static final int BRANCH = 40;     | 转移到指定的存储单元上                     |
| static final int BRANCHNEG = 41;  | 如果累加器为负数，则转移到指定的存储单元上           |
| static final int BRANCHZERO = 42; | 如果累加器为零，则转移到指定的存储单元上            |
| static final int HALT = 43;       | 停止，表示该程序已结束运行                   |

图 5.22 Simpletron 机器语言（SML）的操作码

SML 指令的后两位是操作数，包含要对其应用操作的字所在的存储单元地址。下面，让我们考虑几个简单的 SML 程序。

第一个 SML 程序（如图 5.23 所示）从键盘读取两个数，对其进行计算并打印它们的总和。指令 +1007 从键盘读取第一个数并将其置于单元 07（已初始化为零）。接着，+1008 把第二个数读取到单元 08，load 指令 +2007 将第一个数放入累加器中，add 指令 +3008 将第二个数同累加器中的数相加。所有的算术指令都将其结果存放在累加器中。store 指令 +2109 将结果放入单元 09，write 指令 +1109 取出此数并打印出来（作为一个有符号的 4 位十进制数而打印出来），halt 指令 +4300 终止程序的执行。

| （例 1）单元 | 数字    | 指令     |
|---------|-------|--------|
| 00      | +1007 | （读取 A） |
| 01      | +1008 | （读取 B） |
| 02      | +2007 | （加载 A） |
| 03      | +3008 | （加 B）  |
| 04      | +2109 | （存储 C） |
| 05      | +1109 | （打印 C） |
| 06      | +4300 | （停止）   |
| 07      | +0000 | （变量 A） |
| 08      | +0000 | （变量 B） |
| 09      | +0000 | （结果 C） |

图 5.23 读取并计算两个整数之和的 SML 程序

第二个SML程序(如图5.24所示)从键盘读取两个数,判断并打印较大的值。注意,使用指令+4107作为一个控制转移条件,这很像Java的if语句。现在要求读者编写一些SML程序,以完成下列任务:

- 使用一个条件控制循环来读取10个正数,计算并打印它们的总和。
- 使用一个计数控制循环读取7个数,其中一些是正数,一些是负数,计算并打印它们的平均值。
- 读取一系列的数,判断并打印最大的一个数。第一个读入的数是要处理的数的总数。

| (例2) 单元 | 数     | 指令           |
|---------|-------|--------------|
| 00      | +1009 | (读取A)        |
| 01      | +1010 | (读取B)        |
| 02      | +2009 | (加载A)        |
| 03      | +3110 | (减B)         |
| 04      | +4107 | (如果为负,转移至07) |
| 05      | +1109 | (打印A)        |
| 06      | +4300 | (停止)         |
| 07      | +1110 | (打印B)        |
| 08      | +4300 | (停止)         |
| 09      | +0000 | (变量A)        |
| 10      | +0000 | (变量B)        |

图 5.24 读取两个整数并判断较大值的SML程序

**5.43 (计算机模拟程序)** 在这个练习中,读者将建立自己的计算机,但不是去将具体的元器件组装在一起,而是要使用功能强大、基于软件的模拟技术来创建Simpletron的面向对象的软件模型。Simpletron模拟程序会把我们使用的计算机变成一台Simpletron,使其能实际运行、测试并调试在练习5.42中所编写的SML程序。这个Simpletron将成为一个事件驱动的applet——通过按下按钮来执行每一条SML指令,你可以看到指令将“活动起来”。当运行Simpletron模拟程序时,应当在开始时显示:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time into the input ***
*** text field. I will display the location ***
*** number and a question mark(?). You then ***
*** type the word for that location. Press the ***
*** Done button to stop entering your program. ***
```

这个程序应当显示一个input文本字段,用户可以在该文本字段中一次输入一条指令;并且还应带有一个Done按钮,用户输完SML程序后就按下这个按钮运行。利用一个100元素的单下标数组来模拟Simpletron的存储器,现在假定模拟程序在运行之中,让我们考察当输入练习5.42中例2的程序时将会出现的对话情况:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
```

```

04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000

```

程序应当使用文本字段显示跟在一个问号后面的存储单元。每个问号右边的值由用户输入 input 文本字段。当按下 Done 按钮时，程序应当显示：

```

*** Program loading completed ***
*** Program executing begins ***

```

SML 程序已加载到数组 memory 中。Simpletron 应当提供一个“Execute next instruction”按钮，用户通过按下该按钮来执行 SML 程序的每一条指令。从单元 00 的指令开始，然后顺序进行，除非由于一个控制转移而转向程序的其他部分。

使用变量 accumulator 表示累加器，变量 instructionCounter 则跟踪包含了当前执行指令的存储器单元。变量 operateCode 指明正在执行的指令，即指令字的左边两位，而变量 operand 则指明当前指令正在操作的存储器单元。不要从存储器直接执行指令，而是将要执行的下一条指令传送到一个称为 instructRegister 的变量中，接着“提取”左边两位并将其置于 operand 中。前面的这些寄存器都应有相应的文本字段，可以在任何时候利用文本字段来显示它的当前值。当 Simpletron 模拟程序开始执行时，各种特定的寄存器都将初始化为 0。

现在让我们研究一下第一条 SML 指令，即存储单元 00 处的 +1009。instructionCounter 告诉我们将要执行的下一条指令的单元。我们通过如下语句，从存储器的相应单元中取出其内容：

```
instructionRegister = memory[ instructionCounter ];
```

操作码和操作数可以通过下面的语句而从指令中获取：

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

现在，Simpletron 必须确定操作码是否是一个读（或是写、加载等）操作。使用一个 switch 语句，可以区分出 SML 的 12 种操作。在 switch 结构中，各种 SML 指令的动作模拟如图 5.25 所示。我们将简短地讨论 branch 指令，其余的留给读者自己整理。

| 指令  | 描述                                                                                      |
|-----|-----------------------------------------------------------------------------------------|
| 读取： | 显示提示符“Enter an interger”，使用户能向 input 文本字段输入其值。读取输入的值，将其转换成整数，并存放在单元 memory[ operand ] 中 |
| 加载： | accumulator = memory[ operand ];                                                        |
| 加：  | accumulator += memory[ operand ];                                                       |
| 停止： | 该指令打印如下信息：<br>*** Simpletron execution terminated ***                                   |

图 5.25 Simpletron 中几个 SML 指令的行为

当SML程序执行完后,应该将每个寄存器的名称、内容以及存储器的全部数据显示出来,这样一种打印输出通常称为计算机转储(老式计算机还做不到)。为帮助读者编写方法,图5.26中给出了一个简单的转储格式。注意,执行完Simpletron程序后的转储器将显示停止时刻的指令和数据的值。

```
REGISTERS:
accumulator      +0000
instructionCounter 00
instructionRegister +0000
operationCode      00
operand           00

MEMORY:
      0      1      2      3      4      5      6      7      8      9
00 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

图 5.26 示例转储

在前面给出的转储例子中,假定输出是调用一系列System.out.print和System.out.println方法而显示到显示屏上的。不过,建议使用Graphics的方法drawString或一个TextField对象的数组,试着在applet中显示出结果。

让我们继续第一条指令的执行,即在单元00中的指令+1009。正如我们所指出的,switch语句是通过这样的方法进行模拟的:它提示用户在input文本字段中输入一个值,然后读取这个值,将其转换成整数,再存入单元memory[operand]中。由于Simpletron是事件驱动的,所以它等待用户在input文本字段输入值并按下回车键,该值随后将读入单元09。执行到此,第一条指令的模拟过程就完成了,然后准备执行下一条指令。由于刚才完成的指令不是控制转移,我们将指令计数器加1:

```
++instructionCounter;
```

这样就完成了第一条指令的模拟执行。当用户按下“Execute next instruction”按钮时,就从取出下一条要执行的指令而开始了整个处理过程(即指令执行周期)。现在让我们来考虑如何模拟转移指令,即控制的转移,我们要处理的只是适当调整指令计数器的值。所以,在switch中的无条件转移指令(40)应模拟为:

```
instructionCounter = operand;
```

条件(如果累加器为0则转移)指令应模拟为:

```
if ( accumulator == 0 )
instructionCounter = operand;
```

至此, 我们已经实现了 Simpletron 模拟程序, 并且运行了在练习 5.42 中编写的所有 SML 程序。还可以利用附加的功能来增强 SML, 并将它们添加到模拟程序中。

Simpletron 模拟程序应当检查各种类型的错误。例如, 在程序加载阶段, 用户输入 Simpletron 存储单元的每个数都必须在 -9999 和 +9999 之间。因此模拟程序应当测试每个输入的数是否在此范围内, 如果不是, 就不断提示用户重新输入此数, 直到输入一个正确的数为止。

在执行阶段, 模拟程序也应当检查各种错误, 如出现除数为零、执行非法的操作码、累加器溢出 (即算术操作导致结果大于 +9999 或小于 -9999) 等类似情况。这种严重的错误称为致命错误 (fatal error)。当检测到一个致命错误时, 模拟程序应打印出如下信息:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

并且使用我们讨论过的格式来转储全部的计算机状态, 这将有助于用户在程序中定位错误。

5.44 (Simpletron 模拟程序的改进) 在练习 5.43 中, 我们编写了一个软件模拟程序, 用以执行使用 Simpletron 机器语言 (SML) 编写的程序。在这个练习中, 我们建议对 Simpletron 模拟程序进行修改或增强。在练习 17.26 和练习 17.27 中, 我们将建立一个编译器, 把用高级语言 (由 Basic 演化而来) 编写的程序转换成 Simpletron 机器语言。为了执行编译器生成的程序, 可能需要进行如下一些修正和改进。

- a) 扩展 Simpletron 模拟程序的存储器至 1 000 个存储单元, 使 Simpletron 能够处理更大的程序。
- b) 允许模拟程序执行取模运算, 这需要增加一条 Simpletron 机器语言指令。
- c) 允许模拟程序执行指数运算, 这需要增加一条 Simpletron 机器语言指令。
- d) 修改模拟程序, 使用十六进制值而不是整数值来表示 Simpletron 机器语言指令。
- e) 修改模拟程序, 允许输出一个换行符, 这需要增加一条 Simpletron 机器语言指令。
- f) 修改模拟程序, 使之不但能处理整数, 还能处理浮点数。
- g) 修改模拟程序, 以便处理字符串输入。提示: 每个 Simpletron 字可以分成两组, 每组都包含一个两位整数。每两位整数表示 ASCII 字符的十进制等效形式。增加一条机器指令, 使得模拟程序能够读取输入的字符串, 并从指定单元开始存储这个字符串。存储在此单元首部的半个字表示整个字符串中字符数的总和 (即字符串长度), 随后的每半个字包含一个用两个十进制数字表示的 ASCII 字符。Simpletron 机器语言指令将每个字符转换成等效的 ASCII 字符, 并将其赋给一个“半个字”。
- h) 修改模拟程序, 以处理按照 g) 中的格式进行存储的字符串的输出。提示: 增加一条机器语言指令, 打印从一个指定 Simpletron 存储单元开始的字符串, 存储在此单元的前半个字是字符串中字符的个数 (即字符串的长度), 随后的每半个字包含一个用两位十进制数字表示的 ASCII 字符。机器语言指令检查其长度, 并通过将每个两位数转化成其等效字符来打印这个字符串。

## 第6章 基于对象的编程

### 教学目标

- 理解封装和数据隐藏
- 理解数据抽象的概念和抽象数据类型 (ADT)
- 学会创建Java的ADT (也称为类)
- 学会创建、使用和删除对象
- 学会控制对于对象的实例变量和方法的访问
- 理解面向对象编程的优点
- 理解类变量和类方法
- 理解this引用的使用

### 6.1 简介

现在,我们将在更深的层次上考察Java的面向对象特性。为什么要将这个问题推迟到这一章再进行讨论呢?首先,我们将要建立的对象包含结构化程序段,因此需要具备使用控制结构进行结构化编程的基础;第二,我们想在更深的层次上研究方法;第三,我们想使读者熟悉作为Java对象的数组。

通过第1章到第5章中关于面向对象的Java编程的讨论,我们已经介绍了Java中面向对象编程的许多基本概念(即“对象思想”)和术语(即“对象表述”)。在这些章节中也讨论了程序开发方法学:我们分析了许多典型的、需要用一段程序(要么是一个Java applet,要么是一个Java应用程序)来解决的问题,确定了需要使用哪一种Java API类来实现这个程序,我们的applet和应用程序需要使用什么实例变量和方法,并且指定了类的对象应如何与Java API类的对象相协作以完成程序的总目标。

让我们简要回顾一下面向对象的重要概念和术语。面向对象的编程(OOP)将数据(属性)和方法(行为)封装到对象中,对象的数据和方法紧密地结合在一起。对象具有信息隐藏的性质,这说明尽管对象可能知道如何通过正确定义的接口同其他对象进行通信,但一般不允许对象知道其他对象是如何工作的——实现的细节隐藏在对象的内部。这如同一位驾驶员,即使他不知道汽车引擎、传动器和排气系统的内部工作情况,也可以轻松地驾驶汽车。后面,我们将会解释为什么信息隐藏对于良好的软件工程是如此重要。

在C语言和其他的过程式编程语言中,编程趋向于面向动作(action-oriented);而在Java编程中却是面向对象的。在C语言中,编程的单元是函数(函数在Java中称为方法)。在Java中,编程的单元是类,最终从类中实例化(即创建)对象。函数在Java中并未消失,而是作为方法和要处理的数据一同封装进类中。

C程序员专注于编写函数。完成特定任务的几组动作组成函数,函数最终又组成了程序。在C语言中,数据当然是重要的,但是数据的存在主要是为了支持函数执行的动作。系统需求文档中的

动词帮助C程序员确定将一同工作的、实现系统的函数集合。

Java程序员则专注于创建他们自己的、称为类的用户定义类型。类也称为程序员定义的类型，每个类包含着数据以及操作数据的方法，类中的数据构件称为实例变量（它们在C++中称为数据成员）。就像内置类型（如int）的实例称为变量一样，用户定义类型（即类）的实例称为对象。在Java中，关注的焦点在于对象而非方法。一个系统需求文档中的名词帮助Java程序员决定用于开始设计过程的最初的一组类。然后这些类用于实例化一些对象，它们将一起工作以实现该系统。

本章将解释如何创建和使用对象，即我们通常所说的基于对象的编程（OBP），第7章将介绍继承和多态性——两个真正的OOP的关键技术。

#### 性能提示 6.1

在向方法传递对象时，只是传递对象的引用，而不是复制一个可能很大的对象（就像按值传递的情况一样）。

#### 软件工程视点 6.1

编写易于理解和维护的程序是重要的。改变是规则而非异常，程序员应当随时准备对代码进行修改。正如我们将来看到的，类改善了程序的可修改性。

## 6.2 通过类实现一个抽象数据类型 Time

图6.1包含了对类Time的简单定义。Time类的定义从第3行开始。类定义的结构体使用左右花括号（{和}）括出来。该类定义中包含三个整型实例变量——hour、minute和second。

```

1      // Fig. 6.1: Time.java
2      // Time class definition
3      public class Time {
4          private int hour;           // 0 - 23
5          private int minute;         // 0 - 59
6          private int second;         // 0 - 59
7
8          // Time constructor initializes each instance variable
9          // to zero. Ensures that each Time object starts in a
10         // consistent state.
11         public Time() { setTime( 0, 0, 0 ); }
12
13         // Set a new Time value using military time. Perform
14         // validity checks on the data. Set invalid values
15         // to zero.
16         public void setTime( int h, int m, int s )
17         {
18             hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
19             minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
20             second = ( ( s >= 0 && s < 60 ) ? s : 0 );
21         }
22
23         // Convert Time to String in military-time format
24         public String toMilitaryString()
25         {
26             return ( hour < 10 ? "0" : "" ) + hour +
27                 ( minute < 10 ? "0" : "" ) + minute;
28         }
29
30         // Convert Time to String in standard-time format

```

```

31     public String toString()
32     {
33         return ( ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ) +
34             ":" + ( minute < 10 ? "0" : "" ) + minute +
35             ":" + ( second < 10 ? "0" : "" ) + second +
36             ( hour < 12 ? " AM" : " PM" );
37     }
38 }
39 // Fig. 6.1: TimeTest.java
40 // Class TimeTest to exercise class Time
41 import java.awt.Graphics;
42 import java.applet.Applet;
43
44 public class TimeTest extends Applet {
45     private Time t;
46
47     public void init()
48     {
49         t = new Time();
50     }
51
52     public void paint( Graphics g )
53     {
54         g.drawString( "The initial military time is: " +
55             t.toMilitaryString(), 25, 25 );
56         g.drawString( "The initial standard time is: " +
57             t.toString(), 25, 40 );
58
59         t.setTime( 13, 27, 6 );
60         g.drawString( "Military time after setTime is: " +
61             t.toMilitaryString(), 25, 70 );
62         g.drawString( "Standard time after setTime is: " +
63             t.toString(), 25, 85 );
64
65         t.setTime( 99, 99, 99 );
66         g.drawString( "After attempting invalid settings:",
67             25, 115 );
68         g.drawString( "Military time: " +
69             t.toMilitaryString(), 25, 130 );
70         g.drawString( "Standard time: " + t.toString(),
71             25, 145 );
72     }
73 }

```

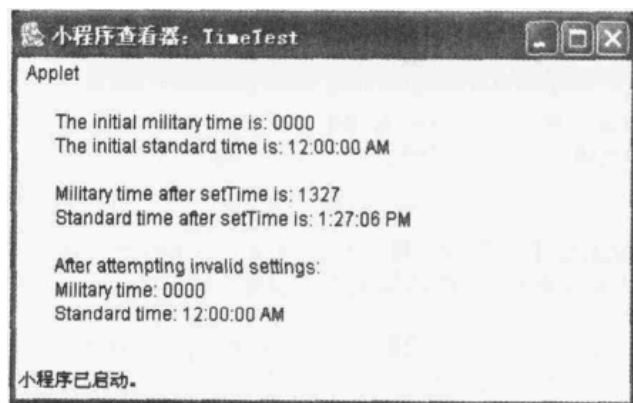


图 6.1 利用类实现抽象数据类型



Time 的 public (公有) 和 private (私有) 关键字称为成员访问修饰符 (member access modifier)。任何使用成员访问修饰符 public 声明的实例变量或方法, 在任何可访问 Time 类的对象的范围内都可以访问; 任何使用成员访问修饰符 private 声明的实例变量或方法则只有本类的方法可以访问。在每一个实例变量和方法的定义前, 必须显式地使用成员访问修饰符。成员访问修饰符在类定义中可以多次出现, 并且可以按照任何顺序出现。

#### 编程技巧 6.1

为了获得程序的清晰性和可读性, 在类定义中应使用成员访问修饰符将成员分组

Time 类包含了以下 public 方法——Time、setTime、toMilitaryString 以及 toString, 这些都是类的 public 方法, 也称为 public 服务或 public 接口。这些方法由类的客户 (即作为使用者的那部分程序) 所使用, 用以操作类的数据

请注意和类具有相同名字的方法, 该方法是类的构造函数 (constructor)。构造函数是一个特殊的方法, 用来初始化一个类对象的实例变量。在创建类的对象时将自动调用该类的构造函数。一个类有几个构造函数是正常的, 这是通过方法重载实现的。构造函数可有参数, 但不能有返回值。

#### 常见编程错误 6.1

试图为构造函数声明返回值或试图从构造函数返回一个值都是语法错误。

三个整型的实例变量 hour、minute 和 second 分别使用 private 成员访问修饰符声明。这表明这个类的三个实例变量仅对类中的方法是可访问的。实例变量一般声明为 private, 而方法一般声明为 public。private 方法和 public 数据也是有可能的, 这一点我们将在后面看到。private 方法通常称为实用方法或帮助方法, 因为它们只能由该类中的其他方法调用, 并用于支持那些方法的操作。使用 public 数据是一种不常见且很危险的编程习惯。

一旦定义了类, 那么该类在声明中就可作为一个类型来使用, 例如:

```
Timesunset,           //reference to object of type Time
TimeArray [ ];        //reference to array of Time objects
```

类名是一个新的类型声明符。一个类可以有几个对象, 就像一个基本数据类型 (如 int) 有许多变量一样。程序员能够根据需要创建新的类, 这是 Java 成为一种可扩展语言的原因之一。

图 6.1 中使用了 Time 类, applet 的 TimeTest 声明了一个称为 t 的 Time 类实例变量, 并将其在 init 方法中初始化。当初始化该对象后, 程序将自动调用 Time 的构造函数并激活 setTime 方法, 显式地将每个私有实例变量初始化为 0。下一步, 自动激活 applet 的 paint 方法, 并且以军用格式 (使用 toMilitaryString 方法) 和标准格式 (使用 toString 方法) 打印时间, 用来确认是否正确地初始化了数据。接着, 使用 setTime 方法将时间设置成另一个正确的值, 并且再一次使用两种格式将其打印出来。然后, setTime 方法将实例变量设置成不合法的值, 并再次用两种格式打印出来。

需要再次注意的是, 实例变量 hour、minute 和 second 都声明为 private。一个类的私有实例变量在类的外部是不可访问的, 这表明类内部使用的实际数据表示同类的客户毫不相干。例如对于类而言, 有充分的理由在类的内部使用从午夜开始的秒数表示时间, 客户可以使用同样的 public 方法获取相同的结果而无需知道这一点。在这种情况下, 类的实现对其客户是隐藏的。练习 6.18 要求对图 6.1 中的 Time 类进行精确的修改, 并且对于类的客户而言, 不会发现相应的改变。

#### 软件工程视点 6.2

信息隐藏增进了程序的可修改性, 简化了客户对类的理解。

### 软件工程视点 6.3

类的客户不必知道一个类实现的内部细节就可以（并且应该）使用该类。如果类的实现改变了（例如为了改进性能），假定类的接口未变，那么类客户的源代码就不需要改动。这样使修改系统更加容易。

在这个程序中，Time 的构造函数简单地将实例变量初始化为 0（即格林威治时间 12 AM）。这确保了创建后的对象处于一个稳定状态——所有的实例变量都是合法的。非法的值不会存储在一个 Time 对象的实例变量中，因为当 Time 对象创建时构造函数已经自动调用了。之后，客户试图修改实例变量的行为都受限于 setTime 方法。

实例变量可以在类的结构体中声明它们的位置上通过类的构造函数来完成初始化，或者通过 set 方法赋值。

### 编程技巧 6.2

在类的构造函数中初始化实例变量

每个类都可能包括一个称为 finalize 的终止函数（finalizer），该函数用于在系统回收对象的存储空间（即该对象成为无用单元）之前完成一些“清理工作”。我们将在后面详细讨论无用单元的回收和终止函数。

有趣的是，toMilitaryString 和 toString 方法都没有参数，因为这些方法隐式地知道它们将要操作的是特定 Time 对象的实例变量，这使得方法调用比过程编程中的传统函数调用更加简单，同时也降低了传递错误参数的可能性，在 C 语言的函数调用中常常发生参数类型错误和参数数目错误。

### 软件工程视点 6.4

使用面向对象的编程方法通常能够减少传递的参数个数，从而简化方法调用。面向对象编程的这一优点来自于这样的事实：在一个对象中封装了实例变量和方法，使方法可以访问实例变量。

类简化了编程，因为客户（或者类对象的使用者）只需关心封装在对象中的操作。这些操作通常设计成面向客户的而不是面向实现的。客户不需要关心一个类的实现，类的接口是会发生改变，但总是低于类的实现的改动频率。当修改类的实现时，必须相应改变与实现相关的代码。通过隐藏实现，我们消除了其他程序部分与类实现相关的可能性。

一般情况下，类的创建并不需要“白手起家”。可以从提供可用操作的那些类中派生出新类，或者可以将其他类的成员作为新类的成员。从已有的类中派生新类的操作称为继承，我们将在第 7 章中详细讨论，包含作为类成员的其他类的对象的操作称为复合或聚集，本章稍后将讨论这个问题。

## 6.3 类作用域

一个类的实例变量和方法属于该类的作用域（scope）。在类的作用域中，类的方法可以直接引用类的成员，只需简单地引用其名字即可。在类的作用域之外，不能直接使用名字来引用类的成员。那些类可见的成员（例如 public 成员）只能通过一个“句柄”来访问，即基本数据类型成员可以使用 objectReferenceName.primitive\_VariableName 的形式来引用，而对象成员可通过 objectReferenceName.objectMemberName 的形式来引用。

在某方法中定义的变量只被该方法所知，即它们对该方法而言是局部变量。可以称这些变量具有块作用域（block scope）。如果方法定义了一个与类作用域的变量（即实例变量）具有相同名字的变量，那么类作用域的变量则由方法作用域的变量所隐藏。在特定方法中，可以通过名字前加上 this 关键字和一个句点来引用被隐藏的实例变量，例如 this.x。

## 6.4 控制对成员的访问

成员访问修饰符 `public` 和 `private` 用于控制对类的实例变量和方法的访问（我们将在第7章介绍另一种访问修饰符 `protected`）。

正如前面所指出的，`public` 方法的主要目的是为类的客户展现类所提供的服务（`service`）的视图，即类的公有接口。类的客户不需要关心类是如何完成其任务的，由于这个原因，类的客户不能访问类的 `private` 方法和 `private` 实例对象（即类的实现细节）。

### 常见编程错误 6.2

非该类成员的方法试图访问该类的一个 `private` 成员将会产生语法错误。

图 6.2 说明了 `private` 类成员不能在外面用名字访问。第 17 行 ~ 第 19 行试图直接访问 `Time` 对象 `t` 的 `private` 实例变量 `hour` 和 `minute`。当编译这个程序时，编译器将产生两个错误，说明每个语句中定义的 `private` 成员是不可访问的。[注意：这个程序假定图 6.1 中的 `Time` 类已被使用。]

```
-----
1      // Fig. 6.2: TimeTest.java
2      // Demonstrate errors resulting from attempts
3      // to access private class members.
4      import java.awt.Graphics;
5      import java.applet.Applet;
6
7      public class TimeTest extends Applet {
8          private Time t;
9
10         public void init()
11         {
12             t = new Time();
13         }
14
15         public void paint( Graphics g )
16         {
17             t.hour = 7;
18
19             g.drawString( "minute = " + t.minute, 25, 25 );
20         }
21     }
-----
```

### 输出结果：

```
TimeTest.java:17: Variable hour in class Time not
                  accessible from class Timetest.
    t.hour = 7;

TimeTest.java:19: Variable minute in class Time not
                  accessible from class TimeTest.
    g.drawString( "minute = " + t.minute, 25, 25 );

2 errors
-----
```

图 6.2 访问一个类的 `private` 成员的错误尝试

### 编程技巧 6.3

一般应在类的首行列出 private 成员

### 编程技巧 6.4

尽管 private 成员和 public 成员可以重复或交叉定义, 但最好将一个类的所有 private 成员放在一个组中, 然后把所有的 public 成员放在另一个组中

### 软件工程视点 6.5

将一个类的所有实例变量设置为 private。如果需要, 提供 public 方法未设置 private 实例变量的值, 或是获取 private 实例变量的值。这一结构有助于对类的客户隐藏其实现细节, 从而减少错误并增进程序的可修改性。

对 private 数据的访问应当由类的方法小心地控制。例如, 为了允许客户读取 private 数据的值, 类可以提供 一个 get (获取) 方法 (也称为访问方法)。为了使客户能够修改 private 数据, 类可以提供 一个 set (设置) 方法 (也称为修改方法)。这种修改似乎破坏了 private 数据的原有概念, 但是 set 方法能提供数据验证的功能 (如范围检查), 以保证合法地设置该值; 同时, set 方法可以在接口中使用的数据格式和实现中使用的数据格式之间进行转换。get 方法无需使用“原始”格式展现数据; 不仅如此, get 方法还能够编辑数据并限制客户将要看到的数据的视图。

### 软件工程视点 6.6

类的设计者使用 private 数据和 public 方法来加强信息隐藏的概念和最小使用权限的原则。如果类的客户需要在类中访问数据, 则应通过该类的 public 方法进行访问, 这样程序员可以控制类的数据的操作 (例如, 数据有效性检查可以防止在对象中存储无效数据)。这就是数据封装的原则。

### 软件工程视点 6.7

类的设计者不需要为每个 private 数据类型提供 set 和 get 方法, 这些功能应当在该方法的确有意义并且由类的设计者仔细考虑之后才提供。

### 测试与调试技巧 6.1

将类的实例变量设置为 private, 或者将类的方法设置为 public, 可以方便调试过程, 因为与数据操作有关的问题都局限于该类的方法中。

## 6.5 实用方法

并非所有的方法都需要定义为 public, 并且作为类的接口的一部分, 一些方法也可以是 private, 并作为类的其他方法的实用方法。

### 软件工程视点 6.8

一般可以将方法分成许多不同的大类: 获取 private 实例变量的方法; 设置 private 实例变量的方法; 实现类的特性的方法; 完成各种机械的琐碎工作的方法, 诸如初始化类的对象, 给类对象赋值, 以及在类和内置类型之间、类和其他类之间进行转换等操作。

访问方法能够读取或显示数据。访问方法的另一个常见用途是测试真假值条件——这些方法往往称为谓词方法 (predicate method), 例如一个适用于任何容器类的 isEmpty 方法。容器类是指能够包含许多对象的类, 例如链表、堆栈或队列 (这些数据结构将在第 17 章和第 18 章深入讨论)。程序可能要在从容器对象中读取一项之前测试 isEmpty; 在程序运行中, 也许要在向容器对象插入另

--项之前测试 isFull。

图 6.3 说明了一个实用方法的概念。实用方法不是类的 public 接口，而是 private 方法，该方法支持类的 public 方法的操作。实用方法并不趋向于由一个类的客户所调用。

SalesPerson 类有一个包含 12 个月销售额的数组。当该数组使用 new 分配内存空间时，这些销售额将由 Java 自动初始化为 0。然后，setSales 方法将数额设置成用户提供的值。public 方法 toString 创建了包含最近 12 个月全部销售额的 String。注意，为了便于使用 toString 方法，这里使用 private 实用方法 totalAnnualSales（第 34 行）汇总了 12 个月的销售额

```

1      // Fig. 6.3: SalesPerson.java
2      // Definition of class SalesPerson with a utility method
3      public class SalesPerson {
4          private double sales[];      // 12 monthly sales figures
5
6          // Constructor method allocates sales array
7          public SalesPerson()
8          {
9              sales = new double[ 12 ];
10         }
11
12         // Method to set one of the 12 monthly sales figures
13         public void setSales( int month, double amount )
14         {
15             if ( month >= 1 && month <= 12 && amount > 0 )
16                 sales[ month - 1 ] = amount;
17         }
18
19         // Private utility method to total annual sales
20         private double totalAnnualSales()
21         {
22             double total = 0.0;
23
24             for ( int i = 0; i < 12; i++ )
25                 total += sales[ i ];
26
27             return total;
28         }
29
30         // Convert the total annual sales figure to a String
31         public String toString()
32         {
33             return "The total annual sales are $" +
34                 totalAnnualSales();
35         }
36     }
37     // Fig. 6.3: SalesPersonTest.java
38     // Demonstrating class SalesPerson and its utility method
39     import java.awt.*;
40     import java.applet.Applet;
41
42     public class SalesPersonTest extends Applet {
43         private SalesPerson s;
44         private double salesFigure;
45         private int month;

```

```

46     private boolean moreInput = true;
47     private String months[] = { "January", "February",
48                                "March", "April", "May", "June",
49                                "July", "August", "September",
50                                "October", "November", "December" };
51
52     // GUI components
53     private Label enterLabel;
54     private TextField currentMonth, enter, total;
55
56     public void init()
57     {
58         s = new SalesPerson();
59         month = 1;
60
61         enterLabel = new Label( "Enter sales for" );
62         currentMonth = new TextField( "January", 10 );
63         currentMonth.setEditable( false );
64         enter = new TextField( 10 );
65         total = new TextField( 30 );
66         total.setEditable( false );
67
68         add( enterLabel );
69         add( currentMonth );
70         add( enter );
71         add( total );
72     }
73
74     public boolean action( Event e, Object o )
75     {
76         if ( e.target == enter ) {
77             if ( moreInput ) {
78                 Double val = Double.valueOf( e.arg.toString() );
79                 s.setSales( month, val.doubleValue() );
80
81                 if ( month == 12 ) {
82                     total.setText( s.toString() );
83
84                     // hide GUI components no longer needed
85                     enterLabel.hide();
86                     currentMonth.hide();
87                     enter.hide();
88                     moreInput = false;
89                 }
90                 else {
91                     month++;
92                     currentMonth.setText( months[ month - 1 ] );
93                     enter.setText( "" ); // clear the text field
94                 }
95             }
96         }
97
98         return true;
99     }
100 }

```



图 6.3 使用一个实用方法类

`SalesPersonTest` 类使用了一个 `SalesPerson` 类的对象以及几个 GUI 构件, 从用户那里获取 12 个月的销售额。当用户提供了一个销售额并在文本字段中按下回车键时, `action` 方法自动激活, 并且将销售额从 `String` 转换成 `Double` 对象。`doubleValue` 方法用于从 `Double` 对象 `val` 中获取 `double` 值。这些值同月份号一同传递给 `setSales` 方法, 用来设置该月的销售额。

如果月份号是 12, 当用户输入了销售额时 (即全年销售额已经输入), 就使用 `SalesPerson` 的 `toString` 方法将结果显示在 `total` 文本字段中。接着, 下面几行:

```
enterLabel.hide ( ) ;  
currentMonth.hide ( ) ;  
enter.hide ( ) ;
```

作为第 81 行的 `if` 语句结构的一部分执行。这些行将对用户隐藏 `enterLabel` 标志、`currentMonth` 文本字段以及 `enter` 文本字段 (就像该程序最后的抓屏结果, 只包含 `total` 文本字段)。隐藏这些元素是为了防止用户继续与程序进行交互。如果月份号不是 12, 则 `action` 方法只是简单地使程序准备接收销售额。通过增加月份号、显示月份名并清除 `enter` 文本字段, 使得用户可以输入下一个销售额。

## 6.6 初始化类对象: 构造函数

当创建对象时, 对象的成员可由构造函数初始化。构造函数是一个与类同名的方法, 程序员提供了构造函数, 每次实例化类的对象时都将自动调用该构造函数。实例变量可以隐式地初始化, 也可以在类的构造函数中初始化, 或者在对象创建之后再设置它们的值。构造函数不能定义返回类型或返回值。可以重载构造函数, 从而为类的初始化提供多种方法。

### 编程技巧 6.5

在适当情况下 (几乎总是如此) 提供一个构造函数, 可以保证每个对象都使用有意义的值进行初始化。

当创建一个类的对象时, 初始化值可以放在类名右侧的括号中。这些初始化值可以作为向类的

构造函数传递的参数，我们将在下一个例子中示例这种操作

如果没有为类定义构造函数，编译器就将创建一个没有任何参数的默认构造函数。类的默认构造函数调用类的构造函数，并使用我们在前面讨论过的方式（即基本数据类型的变量初始化为 0、Boolean 类型初始化为 false，引用类型初始化为 null）来初始化实例变量。程序员也可以提供一个无参构造函数，这一点我们将在下面的例子中看到，如果程序员为一个类定义了构造函数，则 Java 将不再为该创建默认构造函数

#### 常见编程错误 6.3

如果为类提供了构造函数，但是没有提供无参构造函数，同时又试图调用一个无参构造函数来初始化该类的对象，那么就会导致语法错误。可以按无参来调用构造函数，但仅当该类没有构造函数（调用默认构造函数）或者存在无参构造函数时才可以

## 6.7 使用重载的构造函数

一个类的方法可以重载，但只能由同一个类的其他方法重载。为了重载一个类的方法，只需简单地使用相同的名字为方法的每个版本提供一个独立的方法定义。记住，重载的方法必须带有不同的参数表。

#### 常见编程错误 6.4

试图使用具有相同名字和参数的方法去重载一个类的方法是一个语法错误。

图 6.1 中的构造函数将 hour、minute 和 second 初始化为 0（即格林威治时间的 12 AM）。图 6.4 重载了 Time 的构造函数，从而提供了初始化 Time 对象的简便方式，构造函数保证了每个对象创建后就处于一致的状态。在这个程序中，每个构造函数都使用传递到自己的值来调用 setTime 方法，以保证提供给 hour 的值处于 0 到 23 的范围内，保证 minute 和 second 的值各自处于 0 到 59 的范围内。如果某个值超出范围，就由 setTime 方法将其设为 0（这是一个确保实例变量一直处于稳定状态的例子）。第 60 行显示了当使用 new 分配一个 Time 对象时，利用类名后加一组空括号的方式来激活无参构造函数，第 61 行 ~ 第 64 行展示了向 Time 构造函数传递参数的过程。相应的构造函数根据在每个构造函数定义中指定的参数个数、类型和顺序，与在构造函数调用中说明的参数个数、类型和顺序进行匹配，匹配的构造函数将自动调用。

```
1 // Fig. 6.4: Time.java
2 // Time class definition
3 public class Time {
4     private int hour;           // 0 - 23
5     private int minute;        // 0 - 59
6     private int second;        // 0 - 59
7
8     // Time constructor initializes each instance variable
9     // to zero. Ensures that Time object starts in a
10    // consistent state.
11    public Time() { setTime( 0, 0, 0 ); }
12
13    // Time constructor: hour supplied, minute and second
14    // defaulted to 0.
15    public Time( int h ) { setTime( h, 0, 0 ); }
16
```



```

17      // Time constructor: hour and minute supplied, second
18      // defaulted to 0.
19      public Time( int h, int m ) { setTime( h, m, 0 ); }
20
21      // Time constructor: hour, minute and second supplied.
22      public Time( int h, int m, int s ) { setTime( h, m, s ); }
23
24      // Set a new Time value using military time. Perform
25      // validity checks on the data. Set invalid values
26      // to zero.
27      public void setTime( int h, int m, int s )
28      {
29          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
30          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
31          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
32      }
33
34      // Convert Time to String in military-time format
35      public String toMilitaryString()
36      {
37          return ( hour < 10 ? "0" : "" ) + hour +
38                  ( minute < 10 ? "0" : "" ) + minute;
39      }
40
41      // Convert Time to String in standard-time format
42      public String toString()
43      {
44          return ( ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ) +
45                  ":" + ( minute < 10 ? "0" : "" ) + minute +
46                  ":" + ( second < 10 ? "0" : "" ) + second +
47                  ( hour < 12 ? " AM" : " PM" );
48      }
49  }
50  // Fig. 6.4: TimeTest.java
51  // Using overloaded constructors
52  import java.awt.Graphics;
53  import java.applet.Applet;
54
55  public class TimeTest extends Applet {
56      private Time t1, t2, t3, t4, t5;
57
58      public void init()
59      {
60          t1 = new Time();
61          t2 = new Time( 2 );
62          t3 = new Time( 21, 34 );
63          t4 = new Time( 12, 25, 42 );
64          t5 = new Time( 27, 74, 99 );
65      }
66
67      public void paint( Graphics g )
68      {
69          g.drawString( "Constructed with:", 25, 25 );
70          g.drawString( "all arguments defaulted:", 25, 40 );
71          g.drawString( "    " + t1.toMilitaryString(),
72                      25, 55 );

```

```

73      g.drawString( "    " + t1.toString(), 25, 70 );
74
75      g.drawString( "hour specified; minute " +
76                  "and second defaulted:", 25, 85 );
77      g.drawString( "    " + t2.toMilitaryString(),
78                  25, 100 );
79      g.drawString( "    " + t2.toString(), 25, 115 );
80
81      g.drawString( "hour and minute specified; " +
82                  "second defaulted:", 25, 130 );
83      g.drawString( "    " + t3.toMilitaryString(),
84                  25, 145 );
85      g.drawString( "    " + t3.toString(), 25, 160 );
86
87      g.drawString( "hour, minute, and second specified:",
88                  25, 175 );
89      g.drawString( "    " + t4.toMilitaryString(),
90                  25, 190 );
91      g.drawString( "    " + t4.toString(), 25, 205 );
92
93      g.drawString( "all invalid values specified:",
94                  25, 220 );
95      g.drawString( "    " + t5.toMilitaryString(),
96                  25, 235 );
97      g.drawString( "    " + t5.toString(), 25, 250 );
98  }
99  }

```

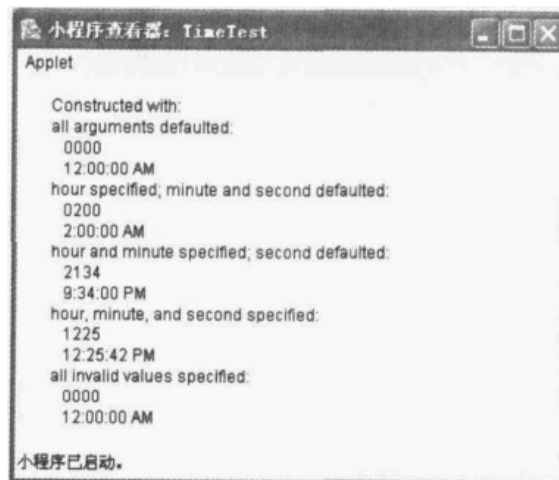


图 6.4 使用重载的构造函数

**编程技巧 6.6**

每一个修改对象 private 实例变量的方法都应保证数据处于稳定状态。

注意,那些没有显式地声明接收 hour、minute 和 second 这三个值的构造函数,如果不满足 setTime 方法对这三个参数的要求,则用 0 值调用 setTime 方法。

注意,每个 Time 构造函数都可以包含 setTime 方法中的适当语句。这也许可以提高程序的效率,因为取消了对 setTime 的额外调用。但是,由于 Time 构造函数和 setTime 方法的代码相类似,因此使程序的维护更加困难。如果 setTime 方法的实现发生了变化,那么 Time 构造函数的实现也需相

应改动。如果直接调用 setTime 的 Time 构造函数需要改动 setTime 的实现，则只需进行一次修改即可，这样就减少了替换实现时发生编程错误的概率。

#### 软件工程视点 6.9

如果类的一个方法已经提供了部分或全部类的构造函数（或其他方法）所需的功能，则让构造函数（或其他方法）去调用这个方法。这样就简化了代码维护，同时减少了由于修改代码实现所带来的错误概率。这也是重用的有效例子。

## 6.8 使用 set 和 get 方法

private 实例变量只能由该类的方法操作。其中一个典型的操作可以是使用 computeInterest 方法来调整消费者银行账户余额（如一个 BankAccount 类的 private 实例变量）。

类经常提供 public 方法，以供类的客户 set（即赋值）或 get（即取值）private 实例变量。这些方法并非一定要称为 set 和 get，但习惯上是这样。更准确地说，一个设置实例变量 interestRate 的方法一般命名为 setInterestRate，而获取 interestRate 的方法一般命名为 getInterestRate，get 方法也称为访问方法或者获取方法。

看来提供 set 和 get 方法的本质同使实例变量为 public 是一样的，这是 Java 语言满足软件工程需求的又一精妙之处。如果实例变量为 public，则实例变量可由程序中的任何方法随意进行读和写。如果实例变量为 private，则 public 类型的 get 方法允许其他方法随意获取数据，但是 get 方法控制了数据的格式和显示结果。public 类型的 set 方法能够（也很可能会）仔细地限制对实例变量值的修改，这样就保证了新的值对这个数据项是合适的。例如，试图将某月的日期设置为 37 的操作将被拒绝，试图将一个人的体重设为负值的操作也将被拒绝。

由于实例变量为 private，因此数据完整性的优点并不是自动实现的——程序员必须提供合法性检查。Java 提供了这样的框架，程序员能够很方便地设计更好的程序。

#### 软件工程视点 6.10

设置 private 数据的方法应当验证特定的新值是否是合适的。如果它们不合适，则 set 方法应当将 private 实例变量放入合适的一致状态中。

类的 set 方法通常返回值，以表明试图对类的某个对象进行非法赋值。这样类的客户可以测试 set 方法的返回值，从而判断其操作的对象是否合法，以及当对象为非法时所采取的相应动作。

图 6.5 扩展了我们的 Time 类，其中包含了 hour、minute 和 second 实例变量的 get 和 set 方法。set 方法严格控制了对实例变量的设置。任何将实例变量设置为不正确值的企图都将导致实例变量被置为 0（从而将实例变量置于一致状态中）。每个 get 方法只是返回相应的实例变量的值。

```
1 // Fig. 6.5: Time.java
2 // Time class definition
3 public class Time {
4     private int hour;           // 0 - 23
5     private int minute;        // 0 - 59
6     private int second;        // 0 - 59
7
8     // Time constructor initializes each instance variable
9     // to zero. Ensures that Time object starts in a
10    // consistent state.
```

```

11     public Time() { setTime( 0, 0, 0 ); }
12
13     // Time constructor: hour supplied, minute and second
14     // defaulted to 0.
15     public Time( int h ) { setTime( h, 0, 0 ); }
16
17     // Time constructor: hour and minute supplied, second
18     // defaulted to 0.
19     public Time( int h, int m ) { setTime( h, m, 0 ); }
20
21     // Time constructor: hour, minute and second supplied.
22     public Time( int h, int m, int s ) { setTime( h, m, s ); }
23
24     // Set Methods
25     // Set a new Time value using military time. Perform
26     // validity checks on the data. Set invalid values
27     // to zero.
28     public void setTime( int h, int m, int s )
29     {
30         setHour( h );    // set the hour
31         setMinute( m ); // set the minute
32         setSecond( s ); // set the second
33     }
34
35     // set the hour
36     public void setHour( int h )
37     { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }
38
39     // set the minute
40     public void setMinute( int m )
41     { minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); }
42
43     // set the second
44     public void setSecond( int s )
45     { second = ( ( s >= 0 && s < 60 ) ? s : 0 ); }
46
47     // Get Methods
48     // get the hour
49     public int getHour() { return hour; }
50
51     // get the minute
52     public int getMinute() { return minute; }
53
54     // get the second
55     public int getSecond() { return second; }
56
57     // Convert Time to String in military-time format
58     public String toMilitaryString()
59     {
60         return ( hour < 10 ? "0" : "" ) + hour +
61             ( minute < 10 ? "0" : "" ) + minute;
62     }
63
64     // Convert Time to String in standard-time format
65     public String toString()
66     {

```

```

67         return ( ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ) +
68             ":" + ( minute < 10 ? "0" : "" ) + minute +
69             ":" + ( second < 10 ? "0" : "" ) + second +
70             ( hour < 12 ? " AM" : " PM" );
71     }
72 }
73 // Fig. 6.5: TimeTest.java
74 // Demonstrating the Time class set and get methods
75 import java.awt.*;
76 import java.applet.Applet;
77
78 public class TimeTest extends Applet {
79     private Time t;
80     private Label hrLabel, minLabel, secLabel;
81     private TextField hrField, minField, secField, display;
82     private Button tickButton;
83
84     public void init()
85     {
86         t = new Time();
87
88         hrLabel = new Label( "Set Hour" );
89         hrField = new TextField( 10 );
90         minLabel = new Label( "Set Minute" );
91         minField = new TextField( 10 );
92         secLabel = new Label( "Set Second" );
93         secField = new TextField( 10 );
94         display = new TextField( 30 );
95         display.setEditable( false );
96         tickButton = new Button( "Add 1 to Second" );
97
98         add( hrLabel );
99         add( hrField );
100        add( minLabel );
101        add( minField );
102        add( secLabel );
103        add( secField );
104        add( display );
105        add( tickButton );
106        updateDisplay();
107    }
108
109    public boolean action( Event e, Object o )
110    {
111        if ( e.target == tickButton )
112            tick();
113        else if ( e.target == hrField ) {
114            t.setHour( Integer.parseInt( e.arg.toString() ) );
115            hrField.setText( "" );
116        }
117        else if ( e.target == minField ) {
118            t.setMinute( Integer.parseInt( e.arg.toString() ) );
119            minField.setText( "" );
120        }
121        else if ( e.target == secField ) {
122            t.setSecond( Integer.parseInt( e.arg.toString() ) );

```

```

123         secField.setText( "" );
124     }
125
126     updateDisplay();
127
128     return true;
129 }
130
131 public void updateDisplay()
132 {
133     display.setText( "Hour: " + t.getHour() +
134         "; Minute: " + t.getMinute() +
135         "; Second: " + t.getSecond() );
136     showStatus( "Standard time is: " + t.toString()+
137         "; Military time is: " + t.toMilitaryString() );
138 }
139
140 public void tick()
141 {
142     t.setSecond( ( t.getSecond() + 1 ) % 60 );
143
144     if ( t.getSecond() == 0 ) {
145         t.setMinute( ( t.getMinute() + 1 ) % 60 );
146
147         if ( t.getMinute() == 0 )
148             t.setHour( ( t.getHour() + 1 ) % 24 );
149     }
150 }
151 }

```

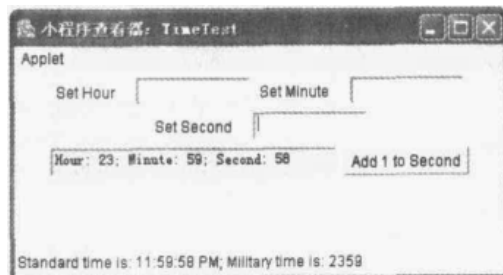
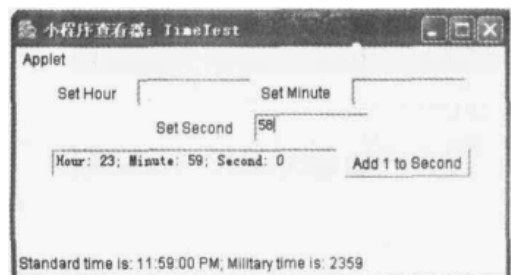
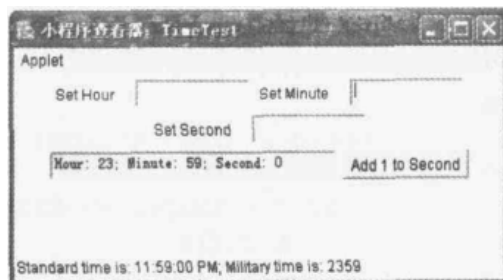
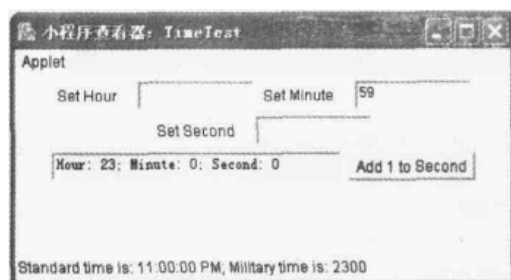
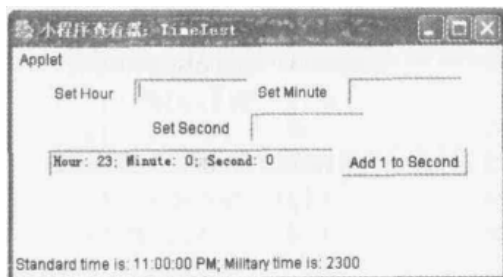
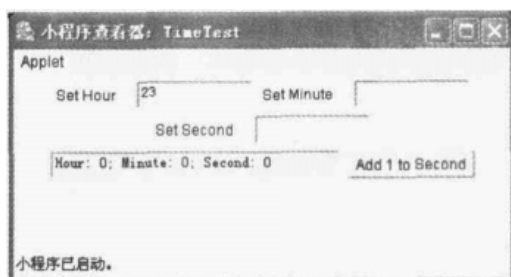




图 6.5 使用 get 和 set 方法

TimeTest applet 提供了一个图形用户界面，使用户可以练习使用 Time 类中的方法。通过在相应的文本字段输入一个值并按下回车键，用户可以设置小时、分钟和秒数，也可以单击“Add 1 to second”按钮来将时间增加 1 秒。在每次操作之后，时间结果都将在 applet 的状态栏中作为一个字符串显示出来。输出窗口中说明了执行下列操作前后 applet 的情况：设置小时为 23、设置分钟为 59、设置秒数为 58，并用“Add 1 to second”按钮两次增加秒数。

注意，当单击“Add 1 to second”按钮时，将调用这个 applet 的 tick 方法。tick 方法使用 set 和 get 方法来增加秒数。尽管这一方式很有效，但它带来了多次方法调用的开销。在下一章中，我们将讨论消除这种性能负担的方法，即友元（friend）访问的概念。

#### 常见编程错误 6.5

构造函数可以调用类的其他方法，诸如 set 或 get 方法，但由于构造函数正在初始化这个对象，因此实例变量可能还未处于一致的状态中。在正确地初始化实例变量之前使用实例变量将产生逻辑错误。

由于 set 方法能够完成合法性检查，因此从软件工程的角度来看，这种方法当然是重要的。set 和 get 方法还具有其他的软件工程方面的优点，下面的“软件工程视点”中将说明这些优点。

#### 软件工程视点 6.11

利用 set 和 get 方法访问 private 数据不仅可以防止实例变量接收非法数值，同时也可以对类的客户隐藏实例变量的内部表示。因此，如果数据的表示发生了变化（一般是为了提高性能而降低所需的存储量），则只需改动方法的实现——客户并不需要改动，前提是方法提供的接口保持不变。

## 6.9 软件可重用性

Java 程序员专注于创建新的类和重用现有的类。许多类库是现成的，而另外一些类库正在世界各地进行开发。因此，软件可以由现有的、定义良好的、精心测试的、可移植的、可以广泛获得的构件组成。这种软件的可重用性加速了功能强大的高质量软件的开发过程，今天，快速应用程序开发（RAD，Rapid Applications Development）已经引起人们的极大兴趣。

为了实现软件可重用性的全部特性，我们需要改进分类设计、注册设计和保护机制，后者可以确保对类的复制不会失败；同时需要改进描述机制，以便系统设计人员确定现有的对象是否满足他们的需要；还要改进浏览机制，从而确定什么类是可获得的以及在多大程度上满足软件开发人员的需求等。许多有趣的研究和开发问题已经解决，但仍有许多问题有待讨论。这些问题最终是会解决，因为软件重用的潜在价值是巨大的。

## 6.10 final 实例变量

我们已经多次强调了最小使用权限的原则 (principle of least privilege), 并将其作为良好的软件工程的基本原则之一。下面, 我们介绍这一原则应用到实例变量上的一种方式。

一些实例变量需要修改而另一些却无需修改。程序员可以使用关键字 `final` 来定义一个变量是不可修改的, 任何对该变量进行的修改都是一个错误。例如, 下列语句:

```
final int increment = 5;
```

声明了一个 `int` 类型的 `final` 实例变量 `increment`, 并将其初始化为 5。

### 软件工程观点 6.12

声明一个 `final` 实例变量有助于增强最小使用权限的原则。如果一个实例变量是不可修改的, 那么就将其声明为 `final` 来表明禁止修改。修改 `final` 实例变量将导致编译错误而非执行错误。

### 常见编程错误 6.6

试图修改 `final` 实例变量将产生语法错误。

图 6.6 的程序创建了一个 `int` 类型的 `final` 实例变量 `increment`, 并将其初始化为 5 (第 8 行)。 `final` 变量不能通过赋值而修改, 必须将其初始化。

```

1 // Fig. 6.6: Increment.java
2 // Initializing a final variable
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class Increment extends Applet {
7     private int count, total;
8     private final int increment = 5; // constant variable
9     private Button incr;
10
11     public void init()
12     {
13         count = 0;
14         total = 0;
15         incr = new Button( "Click to increment" );
16         add( incr );
17     }
18
19     public boolean action( Event e, Object o )
20     {
21         total += increment;
22         count++;
23         showStatus( "After increment " + count +
24                     " : total = " + total );
25         return true;
26     }
27 }
```



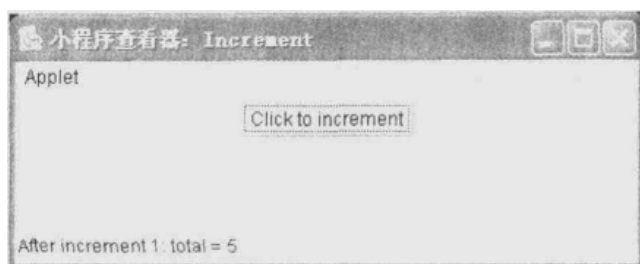


图 6.6 初始化一个 final 变量

如图 6.7 所示，如果图 6.6 中的实例变量声明为 final，而又未在声明时初始化，则将产生编译错误。

```
Increment.java:8: Final variables must be initialized:
                int increment
final int increment; // constant instance variable
1 error
```

图 6.7 未初始化 increment 而导致的编译错误

#### 常见编程错误 6.7

未初始化一个 final 实例变量将产生语法错误。

#### 软件工程视点 6.13

必须初始化常量实例成员（final 对象和 final “变量”），且不允许再对其进行赋值。

## 6.11 复合：作为其他类的实例变量的对象

如果一个 AlarmClock 类对象需要知道何时闹铃，那么为何不将一个 Time 对象作为 AlarmClock 对象的一个成员呢？这种功能称为复合（composition），即一个类可以将其他类的对象作为其成员。

#### 软件工程视点 6.14

复合是一种软件重用形式，它可以使一个类将其他的类对象作为其成员。

图 6.8 中使用 Employee 类和 MyDate 类说明了作为其他对象的成员的对象。Employee 类包含 private 实例变量 firstname、lastname、birthDate 和 hireDate。成员 birthDate 和 hireDate 是 MyDate 类的对象，该类包含 private 实例变量 month、day 和 year。该程序进行的操作是实例化一个 Employee 对象，并初始化和显示它的实例变量。构造函数有 8 个参数（fName、lName、bMonth、bDay、bYear、hMonth、hDay 和 hYear）。参数 bMonth、bDay 和 bYear 将传递给 birthDate 构造函数，参数 hMonth、hDay 和 hYear 将传递给 hireDate 构造函数。

```
1 // Fig. 6.8: MyDate.java
2 // Declaration of the MyDate class.
3 public class MyDate {
4     private int month;           // 1-12
5     private int day;            // 1-31 based on month
6     private int year;           // any year
```

```

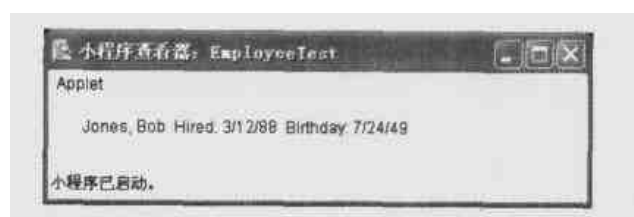
7
8      // Constructor: Confirm proper value for month;
9      // call method function checkDay to confirm proper
10     // value for day.
11     public MyDate( int mn, int dy, int yr )
12     {
13         if ( mn > 0 && mn <= 12 )           // validate the month
14             month = mn;
15         else
16             month = 1;
17         System.out.println( "Month " + mn +
18                             " invalid. Set to month 1." );
19     }
20
21     year = yr;                               // could also check
22     day = checkDay( dy );                     // validate the day
23
24     System.out.println(
25         "MyDate object constructor for date " + toString() );
26 }
27
28 // Utility method to confirm proper day value
29 // based on month and year.
30 private int checkDay( int testDay )
31 {
32     int daysPerMonth[] = { 0, 31, 28, 31, 30,
33                           31, 30, 31, 31, 30,
34                           31, 30, 31 };
35
36     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
37         return testDay;
38
39     if ( month == 2 && // February: Check for leap year
40         testDay == 29 &&
41         ( year % 400 == 0 ||
42           ( year % 4 == 0 && year % 100 != 0 ) ) )
43         return testDay;
44
45     System.out.println( "Day " + testDay +
46                         " invalid. Set to day 1." );
47
48     return 1; // leave object in consistent state
49 }
50
51 // Create a String of the form month/day/year
52 public String toString()
53 { return month + "/" + day + "/" + year; }
54
55 // Fig. 6.8: Employee.java
56 // Declaration of the Employee class.
57 public class Employee {
58     private String firstName;
59     private String lastName;
60     private MyDate birthDate;
61     private MyDate hireDate;
62

```

```

63     public Employee( String fName, String lName,
64                      int bMonth, int bDay, int bYear,
65                      int hMonth, int hDay, int hYear)
66     {
67         firstName = fName;
68         lastName = lName;
69         birthDate = new MyDate( bMonth, bDay, bYear );
70         hireDate = new MyDate( hMonth, hDay, hYear );
71     }
72
73     public String toString()
74     {
75         return lastName + ", " + firstName +
76            "   Hired: " + hireDate.toString() +
77            "   Birthday: " + birthDate.toString();
78     }
79 }
80 // Fig. 6.8: EmployeeTest.java
81 // Demonstrating an object with a member object.
82 import java.awt.Graphics;
83 import java.applet.Applet;
84
85 public class EmployeeTest extends Applet {
86     private Employee e;
87
88     public void init()
89     {
90         e = new Employee( "Bob", "Jones", 7, 24, 49,
91                           3, 12, 88 );
92     }
93
94     public void paint( Graphics g )
95     {
96         g.drawString( e.toString(), 25, 25 );
97     }
98 }

```



输出结果:

```

MyDate object constructor for date 7/24/49
MyDate object constructor for date 3/12/88

```

图 6.8 带有一个成员对象的对象

一个成员对象不需要立即使用构造函数参数来初始化。当创建成员对象时，如果有空参数表，则该对象的默认构造函数（或者无参构造函数）将自动调用。如果含有值，则由默认构造函数（或无参构造函数）创建的值可以随后由 set 方法重置。

**性能提示 6.2**

在构造函数里显式地初始化成员对象，就可以消除两次初始化成员对象的开销：一次是调用成员对象的默认构造函数时，另一次是 set 方法为成员对象提供初始值时。

## 6.12 软件包访问

如果没有给类中定义的方法或者变量提供访问修饰符，那么就认为该方法或变量具有软件包访问性（package access）。如果程序由一个类定义组成，则这一点程序没有特殊影响。但是，如果程序中使用了来自同一个软件包（即一组相关的类）的多个类，那么这些类可以直接通过一个对象的引用来访问其他类的方法和数据。

**性能提示 6.3**

软件包访问使不同类的对象无需使用 set 和 get 方法就能进行交互操作，这样就消除了一些方法调用的开销。

让我们考虑一个软件包访问的例子：图 6.9 中的程序包含 FriendlyDataTest 类和 FriendlyDate 类。在 FriendlyDate 类的定义中，下面两行：

```
int x;           //friendly instance variable
String s;        //friendly instance variable
```

没有使用成员访问修饰符声明来实例变量 x 和 s，因此这些变量是友元（friend）实例变量。FriendlyDateTest applet 创建了一个 FriendlyDate 的实例，以此展示直接修改 FriendlyDate 实例变量的功能（见第 21 行和第 22 行）。修改的结果可以在输出窗口中看到。

注意，当编译这个程序时，产生了两个分离的文件：其中一个 Class 文件是 FriendlyDate 类的，另一个 Class 文件是 FriendlyDateTest 类的，这两个文件自动存储在同一个目录下并将其视为同一个软件包（由于它们处在一个程序中，因此是相关的）的一部分。由于它们是同一个软件包的一部分，因此允许 FriendlyDateTest 类修改 FriendlyDate 类对象的软件包访问的数据。

**软件工程视点 6.15**

OOP 领域的一些观点认为软件包访问破坏了信息隐藏，降低了面向对象设计方法的价值。

```
1 // Fig. 6.9: FriendlyDataTest.java
2 // Classes in the same package (i.e., the same directory)
3 // can access friendly data of other classes in the
4 // same package.
5 import java.awt.Graphics;
6 import java.applet.Applet;
7
8 public class FriendlyDataTest extends Applet {
9     private FriendlyDate d;
10
11     public void init()
12     {
13         d = new FriendlyDate();
14     }
15
16     public void paint( Graphics g )
```

```
17     {
18         g.drawString( "After instantiation: ", 25, 25 );
19         g.drawString( d.toString(), 40, 40 );
20
21         d.x = 77;
22         d.s = new String( "Good bye" );
23         g.drawString( "After changing values: ", 25, 55 );
24         g.drawString( d.toString(), 40, 70 );
25     }
26 }
27
28 class FriendlyData {
29     int x;           // friendly instance variable
30     String s;        // friendly instance variable
31
32     // constructor
33     public FriendlyData()
34     {
35         x = 0;
36         s = new String( "Hello" );
37     }
38
39     public String toString()
40     {
41         return "x: " + x + "      s: " + s;
42     }
43 }
```

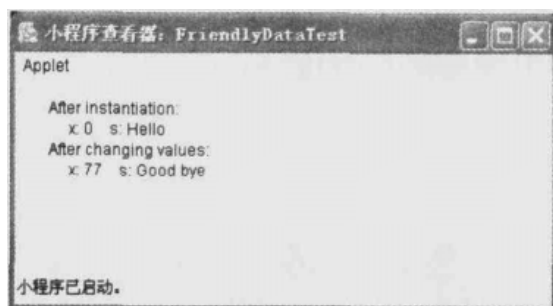


图 6.9 以友元方式访问一个类的成员

## 6.13 使用 this 引用

当一个类的方法由于某种特殊目的而引用该类的另一个成员时,Java如何保证引用合适的对象呢?答案是每个对象都可以访问其自身的引用——称为 `this` 引用。

`this` 引用不是存储在对象自身内的实例数据。相反,它是作为每个非静态方法调用时的隐含参数而传递到对象内部的(我们将在6.15节讨论静态方法)。因此,当一个对象的任何非静态方法运行时,这个方法就可以访问 `this` 引用。

### 常见编程错误 6.8

在一个静态方法中显式地使用 `this` 引用将产生语法错误。

this 引用用于隐式地引用一个对象的实例变量和方法。现在我们给出一个显式使用 this 引用的简单例子，稍后我们给出一些使用 this 特性的例子。

### 性能提示 6.3

Java 通过维护每个类中各个方法的一份副本来保留存储信息，这个方法将由该类的每个对象调用。另一方面，每个对象都带有自己的一份类的实例变量的副本。

图 6.10 展示了显式使用 this 引用，以确保 ThisTest 类的一个方法打印出 ThisTest 对象的一个 private 数据 x。

图 6.10 中的 toString 方法组成一个含有值 x 的字符串。首先由 toString 直接访问 x 的值，然后 toString 通过 this 引用和点运算符 (.) 引用 x。

```

1 // Fig. 6.10: ThisTest.java
2 // Using the this reference to refer to
3 // instance variables and methods.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class ThisTest extends Applet {
8     private int x = 12;
9
10    public void paint( Graphics g )
11    {
12        g.drawString( this.toString(), 25, 25 );
13    }
14
15    public String toString()
16    {
17        return "x: " + x + "    this.x = " + this.x;
18    }
19 }
```

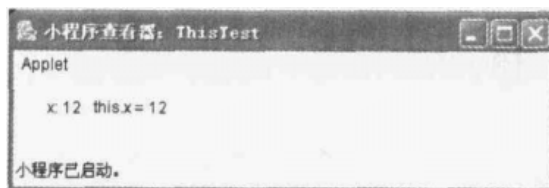


图 6.10 使用 this 引用

另一个 this 引用的使用方式是在激活连接方法调用 (concatenated method calls; 也称为级联方法调用, cascaded method calls) 中。图 6.11 示例了返回一个 Time 对象的引用，使得可以连接 Time 类的方法调用。setTime、setHour、setMinute 和 setSecond 方法各自利用一个 Time 返回类型来返回 this 引用。

```

1 // Fig. 6.11: Time.java
2 // Time class definition
3 public class Time {
4     private int hour; // 0 - 23
5     private int minute; // 0 - 59
6     private int second; // 0 - 59
```

```
7
8    // Time constructor initializes each instance variable
9    // to zero. Ensures that Time object starts in a
10   // consistent state.
11   public Time() { setTime( 0, 0, 0 ); }
12
13   // Time constructor: hour supplied, minute and second
14   // defaulted to 0.
15   public Time( int h ) { setTime( h, 0, 0 ); }
16
17   // Time constructor: hour and minute supplied, second
18   // defaulted to 0.
19   public Time( int h, int m ) { setTime( h, m, 0 ); }
20
21   // Time constructor: hour, minute and second supplied.
22   public Time( int h, int m, int s ) { setTime( h, m, s ); }
23
24   // Set Methods
25   // Set a new Time value using military time. Perform
26   // validity checks on the data. Set invalid values
27   // to zero.
28   public Time setTime( int h, int m, int s )
29   {
30       setHour( h ); // set the hour
31       setMinute( m ); // set the minute
32       setSecond( s ); // set the second
33
34       return this; // enables chaining
35   }
36
37   // set the hour
38   public Time setHour( int h )
39   {
40       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
41
42       return this; // enables chaining
43   }
44
45   // set the minute
46   public Time setMinute( int m )
47   {
48       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
49
50       return this; // enables chaining
51   }
52
53   // set the second
54   public Time setSecond( int s )
55   {
56       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
57
58       return this; // enables chaining
59   }
60
61   // Get Methods
62   // get the hour
63   public int getHour() { return hour; }
64
65   // get the minute
```

```

66     public int getMinute() { return minute; }
67
68     // get the second
69     public int getSecond() { return second; }
70
71     // Convert Time to String in military-time format
72     public String toMilitaryString()
73     {
74         return ( hour < 10 ? "0" : "" ) + hour +
75             ( minute < 10 ? "0" : "" ) + minute;
76     }
77
78     // Convert Time to String in standard-time format
79     public String toString()
80     {
81         return ( ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ) +
82             ":" + ( minute < 10 ? "0" : "" ) + minute +
83             ":" + ( second < 10 ? "0" : "" ) + second +
84             ( hour < 12 ? " AM" : " PM" );
85     }
86 }
87 // Fig. 6.11: TimeTest.java
88 // Chaining method calls together with the this reference
89 import java.awt.Graphics;
90 import java.applet.Applet;
91
92 public class TimeTest extends Applet {
93     private Time t;
94
95     public void init()
96     {
97         t = new Time();
98     }
99
100    public void paint( Graphics g )
101    {
102        t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
103        g.drawString( "Military time: " +
104            t.toMilitaryString(), 25, 25 );
105        g.drawString( "Standard time: " + t.toString(),
106            25, 40 );
107
108        g.drawString( "New standard time: " +
109            t.setTime( 20, 20, 20 ).toString(), 25, 70 );
110    }
111 }

```



图 6.11 连接的方法调用



为什么返回 `this` 引用的技术是如此有效呢？点运算符 (.) 从左至右相结合，于是表达式：

```
t.setHour (18).setMinute (30).setSecond (22);
```

首先计算 `t.setHour(18)`，然后返回作为这一方法调用值的对象 `t` 的引用。表达式的剩余部分解释为：

```
t.setMinute (30).setSecond (22);
```

`setMinute(30)`调用执行并返回对 `t` 的引用。表达式的剩余部分解释为：

```
t.setSecond (22);
```

注意，下列调用：

```
t.setTime (20 , 20 , 20 ).toString ();
```

也使用了连接特征。这些方法调用必须以上述顺序出现在表达式中，因为在类中定义的 `toString` 没有返回对 `t` 的引用。将 `toString` 放在 `setTime` 之前将产生语法错误。

参见编程错误 6.9

在一个方法中，如果某个方法的参数与类的某个成员同名，那么为了访问该类成员，需要显式地使用 `this` 引用；否则，就会错误地引用方法的参数。

**编程技巧 6.7**

避免使用和类成员名相冲突的方法参数名

**编程技巧 6.8**

在某些可以使用 `this` 引用的上下文中，显式地使用 `this` 引用可以增加程序的清晰度。

## 6.14 终止函数

我们已经看到在创建类时，构造函数能够在—个类的对象内初始化数据。构造函数常常需要各种系统资源，例如内存（当使用 `new` 运算符时）。我们需要一种规范的方式将不再需要的资源返回给系统，以避免资源泄漏。构造函数获得的常见的资源就是内存。但是在 Java 中我们无需担心是否需要向系统返回内存，因为 Java 实现了内存无用单元的自动回收。因此，在其他语言（像 C 和 C++）中经常出现的内存泄漏问题，在 Java 中则并不会出现。但是，其他资源泄漏仍然会发生。

Java 中的每个类都有一个终止函数（`finalizer`），该函数可以将资源返回给系统。对象的终止函数可以保证在 Java 回收该对象之前完成清理工作。类的终止函数通常命名为 `finalize`，终止函数没有参数，并且不返回值（即它的返回类型为 `void`）。一个类只可以有一个 `finalize` 方法——不允许重载终止函数。

到目前为止，我们还没有为所提及的类提供终止函数。实际上，简单的类很少用到终止函数。我们将在图 6.12 中看到一个简单的 `finalize`。

## 6.15 静态类成员

类的每个对象都带有类中所有实例变量的副本。在特定条件下，某个变量应该只有一份副本由类的所有对象共享。`static`（静态）类变量表示“类范围内”的信息，`static` 成员的声明由关键字 `static` 开始。

让我们通过一个视频游戏的例子来满足对 static 的类范围数据的需求。假设有一个视频游戏，在游戏中有 Martian（火星星人）和其他太空生物。当 Martian 意识到目前至少有 5 个 Martian 的时候，每个 Martian 都将很勇敢地攻击其他太空生物。如果少于 5 个 Martian，则所有 Martian 就停止攻击，因此每个 Martian 都需知道 martianCount。我们可以将 martianCount 作为实例数据赋给 Martian 类，因此每个 Martian 将有一份单独的实例数据副本。在每次创建一个新的 Martian 时，我们不得不在每个 Martian 中更新实例变量 martianCount。这会由于冗余副本而浪费了内存空间，也会由于更新单独的副本而浪费了处理时间。相反，我们将 martianCount 声明为 static，这样就使 martianCount 成为类范围数据。每个 Martian 都将 martianCount 作为自己的实例数据，然而 Java 只维护静态数据 martianCount 的一份副本，这样就节省了内存空间。我们通过 Martian 构造函数将 martianCount 加 1 来节约处理时间。因为只存在一份副本，因此不再需要为每个 Martian 对象增加 martianCount 的单独副本。

#### 性能提示 6.4

当数据只需一份副本时，将其声明为 static 类变量来节省内存空间。

尽管 static 类变量可能看起来更像全局变量，但 static 类变量仍然具有类作用域。一个类的 public static 类成员可以由该类的任意对象访问，或者它们也可通过类名后加点运算符来访问。一个类的 private static 类成员可通过类的方法访问。实际上，即使那个类中没有对象，static 类成员仍然存在。当类的对象都不存在时，为了访问 public static 类成员，只需在类成员前加上类名和点运算符即可。当类的对象都不存在时，为了访问 private static 类成员，必须提供一个 public static 方法，并且方法调用必须加上类名和点运算符。

图 6.12 的程序展示了一个 private static 实例变量和一个 public static 方法的使用。实例变量 count 默认初始化为零。实例变量 count 表示 Employee 类已经实例化的对象的个数。当 Employee 类的对象存在时，成员 count 可以通过一个 Employee 对象的任何方法访问——在本例中，通过构造函数来引用 count。当 Employee 类没有对象存在时，仍然可以引用成员 count，但只能通过对 static 方法 getCount 的以下调用来实现：

```
Employee.getCount();
```

在这个例子中，getCount 方法用于确定当前已实例化的 Employee 的对象个数。注意，当程序中无实例化的对象时，激活 Employee.getCount() 方法调用。但是，当存在实例化的对象时，getCount 方法也能通过某一个对象来调用，如下所示：

```
e1.getCount();
```

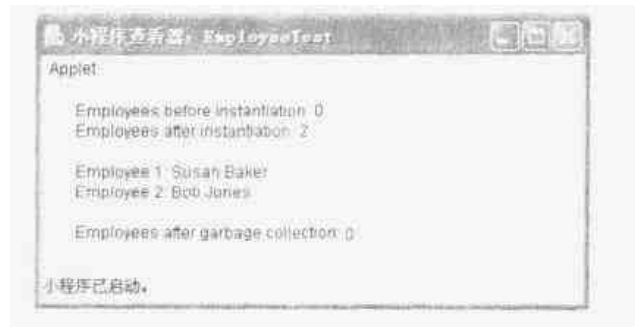
---

```

1      // Fig. 6.12: Employee.java
2      // Declaration of the Employee class.
3      public class Employee {
4          // Instance variables
5          private String firstName;
6          private String lastName;
7          private static int count; // # of objects instantiated
8
9          public Employee( String fName, String lName )
10         {
11             firstName = fName;
12             lastName = lName;

```

```
12
13         ++count; // increment static count of employees
14         System.out.println( "Employee object constructor: " +
15                             firstName + " " + lastName );
16     }
17
18     public void finalize()
19     {
20         --count; // decrement static count of employees
21         System.out.println( "Employee object finalizer: " +
22                             firstName + " " + lastName );
23     }
24
25     public String getFirstName()
26     { return new String( firstName ); }
27
28     public String getLastName()
29     { return new String( lastName ); }
30
31     public static int getCount() { return count; }
32 }
33 // Fig. 6.12: EmployeeTest.java
34 // Test Employee class with static class variable,
35 // static class method, and dynamic memory.
36 import java.awt.Graphics;
37 import java.applet.Applet;
38
39 public class EmployeeTest extends Applet {
40     public void paint( Graphics g )
41     {
42         g.drawString( "Employees before instantiation: " +
43                     Employee.getCount(), 25, 25 );
44         Employee e1 = new Employee( "Susan", "Baker" );
45         Employee e2 = new Employee( "Bob", "Jones" );
46
47         g.drawString( "Employees after instantiation: " +
48                     e1.getCount(), 25, 40 );
49
50         g.drawString( "Employee 1: " + e1.getFirstName() +
51                     " " + e1.getLastName(), 25, 70 );
52         g.drawString( "Employee 2: " + e2.getFirstName() +
53                     " " + e2.getLastName(), 25, 85 );
54
55         e1 = null;
56         e2 = null;
57
58         System.gc(); // explicit call to garbage collector
59
60         g.drawString( "Employees after garbage collection: " +
61                     Employee.getCount(), 25, 115 );
62     }
63 }
```



输出结果:

```
Employee object constructor: Susan Baker
Employee object constructor: Bob Jones
Employee object finalizer: Susan Baker
Employee object finalizer: Bob Jones
```

图 6.12 使用 static 类变量来维护类的对象个数

注意, Employee 类有一个 finalize 方法, 这个方法显示它何时被调用。applet Employee 中的方法 paint 将声明、实例化并使用两个 Employee 对象。当 paint 方法使用这些对象完成其处理时, 将引用 e1 和 e2 置为 null, 即它们不再引用第 44 行和第 45 行实例化的对象。这样就对作为无用单元回收的对象进行了标记, 因为在程序中再没有引用指向这些对象。最后, 无用单元回收器回收这些对象的内存 (或者在程序终止时由操作系统回收)。因为无法确定无用单元回收器何时工作, 所以我们使用以下语句显式地调用它:

```
System.gc (); //explicit call to garbage collector
```

这条语句激活了 System 类的 static 方法 gc, 以安排无用单元回收器尽可能快地工作 (在本程序中, 它在 paint 方法结束执行时开始工作)。我们为这一程序提供了两个输出, 抓屏结果显示了 Employee 对象的使用。第二个输出显示了调用两个 Employee 对象的 finalize 方法, 这表明无用单元回收器从内存中回收了这些对象。

使用 static 声明的方法不能访问非静态成员。与非静态方法不同的是, 静态方法没有 this 引用, 因为 static 类变量和 static 类方法的存在不依赖于一个类的任何对象。

#### 常见编程错误 6.10

在 static 方法中进行 this 引用将产生语法错误。

#### 常见编程错误 6.11

使用 static 方法来调用实例方法或访问实例变量将产生语法错误。

#### 软件工程视点 6.16

即使在一个类中还没有实例化任何对象, static 类变量和 static 类方法也已存在并可以使用。

## 6.16 数据抽象和信息隐藏

在正常情况下, 类向其客户隐藏它们的实现细节, 这就是所谓的信息隐藏 (information hiding)。作为信息隐藏的例子, 我们考虑一个称为堆栈 (stack) 的数据结构。

不妨用一堆盘子来比喻一个堆栈。当把一只盘子放在堆中时, 通常是放在顶部 (称为向堆栈中

压入),当一只盘子从堆上移去时,通常是从顶部移走(称为从堆栈中弹出)。堆栈是后进先出(LIFO)的数据结构,在堆栈中最后压入(插入)的一项最先从堆栈中弹出(移走)。

程序员可以创建一个堆栈,并向客户隐藏它的实现细节。很容易用数组或其他方法(如链表,详细内容请参见第17章和第18章)来实现堆栈。一个堆栈类的客户不需要知道堆栈是如何实现的;客户仅需知道,当数据放置在堆栈中时它们将按后进先出的顺序调用。这一概念称为数据抽象。Java的类定义了抽象数据类型(ADT)。尽管用户也许想知道一个类的实现细节,但用户不会写出依赖于这些细节的代码。这就说明只要这个类的公有接口没有改动,那么一个特定的类(诸如实现一个堆栈及其压入和弹出操作的类)可以由另一个版本替代,而不会影响系统的其他部分。

高级语言的任务就是为程序员的使用来创造一个方便的操作环境,但是没有惟一可接受的标准观点——这就是为什么有如此众多的编程语言的原因之一。Java的面向对象编程也提出了另外一种观点。

大多数编程语言强调动作,在这些语言中,数据的存在是为了支持程序所要采取的动作。数据比起动作来说是要“逊色”一些的,数据是“原始的”,一般只存在少数几种内置的数据类型,程序员难以创建他们自己的新的数据类型。

随着Java和面向对象风格的编程的出现,这种情况正在改变,Java重视数据的重要性。Java的主要行为就是创建新的数据类型(即类),并且表达那些数据类型的对象间的交互作用。

为了在这个方向上可以有进一步的深入,编程语言需要形式化数据的某些概念,我们考虑的形式化就是抽象数据类型(ADT)的概念。今天ADT所引起的关注,就如同20年前结构化编程所引起的关注一样。ADT并不会取代结构化编程;相反,它们提供了其他的形式化来进一步改进程序的开发过程。

什么是一个抽象数据类型?考虑内置类型int,我们首先想到的是数学中整数的概念;但在计算机中,int并不完全是数学中的整数。进一步来说,计算机的int类型数据通常在大小上非常有限。例如,在一个32位机上,int类型大约限制在-20亿到+20亿之间。如果一个计算在这个范围之外,就会出现错误,机器会以某种与机器有关的方式做出响应,包括产生错误。数学中的整数却无此问题。因此,一个计算机的int类型的概念,实际上只是对真实世界中整数概念的近似。对于float类型,情况也是一样。

其中的关键问题在于,甚至在Java这类编程语言中提供的内置数据类型,实际上也只是真实世界中的概念和行为的近似或模型。在此之前,我们想当然地使用int类型,但现在却有了新的认识。如同int、float、char和其他类型都是抽象数据类型的例子,它们是将真实世界的概念表达达到计算机中某种可满意水平的基本方式。

抽象数据类型实际上有两个概念,分别称为数据表示(data representation)和允许对该数据进行的操作(operation)。例如,int的概念定义了Java中的加、减、乘、除和取模运算,但是没有定义除数为0的操作。另一个例子是负整数的概念,它的操作和数据表示是很清晰的,但是对负整数求平方根却是无定义的。在Java中,程序员利用类来实现抽象数据类型。

Java有一个小的基本类型集合,ADT扩展了基本的编程语言。

#### 软件工程视点 6.17

程序员能够通过类机制的使用来创建新类型,这些新类型可以同内置类型一样方便地使用。因此,Java是一个可扩展语言。尽管该语言使用的这些新类型是易于扩展的,但基本语言本身却是不可改变的。。

新的Java类可以面向个人、小组、团体等,许多类都放在标准类库(class library)中,可以广

泛地使用。尽管基于实际的标准正在形成,但这并不表示能够推进标准的形成。只有当基本的、标准化的类库获得比今天更广泛的应用时,Java的全部价值才能得以实现。在美国,这类标准化工作常常是通过ANSI(即美国国家标准委员会)进行的。不管这些库最终以何种形式出现,学习Java和面向对象编程的读者一定可以充分利用类库带来的一种全新的、快速的、面向组件的软件开发方法。

### 6.16.1 案例:队列抽象数据类型

我们中的每个人每时每刻都处在某条等待线上。等待线也称为队列(queue),例如我们在线上等待超市的收银机,我们在线上等待汽车加油,我们在线上等待公共汽车,我们在线上等待交付高速公路费用;计算机系统也在内部使用等待线,于是,我们需要编写程序来模拟队列的属性和行为。

队列是抽象数据类型的很好的例子,一个队列提供给它的客户众所周知的操作。客户每次在队列中放置相应的内容,即使用一个入队操作,然后根据需要依次取回这些内容,即使用一个出队操作。结果,一个队列能变得无限长。从一个队列中返回的项是基于先入先出(FIFO)顺序的——第一个插入到队列中的项将首先从队列中删除。

队列隐藏了当前等在线上的各数据项的内部数据表示;同时它也向其客户提供一组操作,称为入队和出队。客户们并不关心队列的实现,只要队列的操作同其“声称的”一样即可。当插入新的项时,队列应当接收这个项,并以某种先入先出的数据结构在内部安置该项。当客户想要从队列头中取得下一个项时,队列应当从内部表示中删去此项,并且按FIFO的顺序传递给外界,下一个出队操作所返回的项应该是队列中等待时间最长的一项。

队列ADT保证了其内部数据结构的完整性。客户不可以直接操作这个数据结构。只有队列ADT可以访问它的内部数据。客户只可以进行允许的操作,例如在数据表示上执行操作。ADT的公有接口未提供的操作会由ADT以某种适当的方式拒绝,这可能是发出一条错误消息,终止执行,或只是忽略操作请求。

## 小结

- 类的成员通过成员访问运算符——句点运算符(.)来访问。
- 类使程序员能使用属性和行为来模拟对象。类的类型在Java中使用关键字class定义。
- 类定义以关键字class开始,以分号结束,定义的结构体使用花括号({和})括起。
- 任何类中的public实例变量或方法,对任何访问该类对象的方法都是可见的。
- 任何使用private声明的实例变量或方法仅对该类的其他成员可见。
- 在类定义中,成员访问修饰符可以按照任意顺序多次出现。
- 构造函数是与类同名的特殊方法,用于初始化类对象中的成员。构造函数在实例化类的对象时调用。
- 类的public方法集合称为类的公有接口。
- 调用方法比过程编程中的调用函数更加精简,因为方法所用的大多数数据已经在对象中了。
- 在类的作用域中,只能通过名字来访问类的成员。在类的作用域之外,可以通过对象的引用来访问类的成员。
- 公有类成员为类的客户提供了服务接口。
- 为了使客户能够读取private数据,类可以提供get方法。为了使客户能够修改private数据,

类可以提供 set 方法。

- 一般情况下, 类的实例变量为 private 类型, 而方法为 public 类型。一些方法可能为 private 类型, 从而成为该类其他方法的实用方法。
- 可以重载构造函数。
- 一旦正确地初始化了类对象, 所有操作该对象的方法都应确保对象处于一个稳定的状态中。
- 当声明类的对象时, 可以提供初始化值, 这些初始化值都传递给类的构造函数。
- 构造函数不应该标明返回类型, 而且也不应该试图返回值。
- 在系统收集对象的存储空间之前, 使用终止函数执行相应的清理工作。
- 关键字 final 定义的对象是不可修改的。
- final 对象必须在声明时初始化。
- 其他类的对象可以复合为新的类。
- this 引用隐式地用于引用对象内的方法和实例变量。
- new 运算符自动创建适当大小的对象, 并返回一个正确类型的引用。
- 静态类变量表达了“类范围”的信息, 静态成员的声明以关键字 static 开始。
- 静态类变量具有类作用域。
- static 方法不能访问非静态类成员, 并且 static 方法没有 this 引用, 因为 static 类变量和 static 方法不依赖一个类的任何对象而存在。

## 术语

|                                                       |                                          |
|-------------------------------------------------------|------------------------------------------|
| abstract data type(ADT) 抽象数据类型 (ADT)                  | default constructor 默认构造函数               |
| access method 访问方法                                    | dot operator(.) 句点运算符 (.)                |
| aggregation 聚集                                        | encapsulation 封装                         |
| attribute 属性                                          | extends 扩展                               |
| behavior 行为                                           | extensibility 可扩展性                       |
| cascaded method calls 级联的方法调用                         | finalizer 终止函数                           |
| class 类                                               | “friendly” access 友好地访问                  |
| class definition 类定义                                  | get method get (获取) 方法                   |
| class library 类库                                      | helper method 帮助方法                       |
| class method(static) 类方法 (静态)                         | implementation of a class 类的实现           |
| class scope 类作用域                                      | information hiding 信息隐藏                  |
| class variable 类变量                                    | initialize a class object 初始化类对象         |
| client of a class 类的客户                                | instance method 实例方法                     |
| composition 复合                                        | instance of class 类的实例                   |
| concatenated method calls 连接的方法调用                     | instance variable 实例变量                   |
| consistent state for an instance variable 一个实例变量的稳定状态 | instantiate an object of a class 实例化类的对象 |
| constructor 构造函数                                      | interface to a class 类的接口                |
| container class 容器类                                   | member access control 成员访问控制             |
| data type 数据类型                                        | member access operator(.) 成员访问运算符 (.)    |
|                                                       | member access modifier 成员访问修饰符           |

|                                                |                                                 |
|------------------------------------------------|-------------------------------------------------|
| message 消息                                     | programmer-defined type 程序员定义的类型                |
| method 方法                                      | public                                          |
| method calls 方法调用                              | public interface of a class 类的公有接口              |
| mutator method 变更方法                            | query method 查询方法                               |
| new operator new 运算符                           | rapid application development(RAD) 快速应用开发 (RAD) |
| no-argument constructor 无参构造函数                 | reusable code 可重用的代码                            |
| object 对象                                      | service of a class 类的服务                         |
| object-based programming(OBP) 基于对象的编程 (OBP)    | set method set (设置) 方法                          |
| object-oriented programming(OOP) 面向对象的编程 (OOP) | software reusability 软件的可重用性                    |
| package access 软件包访问                           | static class variable 静态类变量                     |
| predicate method 谓词方法                          | static method 静态方法                              |
| principle of least privilege 最小使用权限的原则         | this method this 方法                             |
| private                                        | user-defined type 用户定义的类型                       |
|                                                | utility method 实用方法                             |

## 自测练习

### 6.1 填空:

- 类成员通过 \_\_\_\_\_ 运算符来访问类的对象
- 标记成 \_\_\_\_\_ 的类成员只能由该类的方法访问。
- \_\_\_\_\_ 是一个特殊的方法, 用于初始化类的实例变量。
- \_\_\_\_\_ 方法用于给类的 private 实例变量赋值。
- 正常情况下, 类的方法是 \_\_\_\_\_ 类型, 而实例变量是 \_\_\_\_\_ 类型。
- \_\_\_\_\_ 方法用于接收类的 private 数据。
- 关键字 \_\_\_\_\_ 引入类的定义。
- 当类的成员标记成 \_\_\_\_\_ 时, 处于其作用域的类的对象都是可访问的。
- \_\_\_\_\_ 运算符动态地为一个特定类型分配空间并返回该类型的 \_\_\_\_\_。
- 一个 \_\_\_\_\_ 实例变量表达了类范围的信息。
- \_\_\_\_\_ 关键字规定了一个对象或变量在其初始化之后是不能修改的。
- 一个声明为 static 的方法不能访问 \_\_\_\_\_ 类成员。

## 自测练习答案

- 6.1 a) 句点 (.). b) private. c) 构造函数。d) set. e) public, private. f) get. g) class. h) public. i) new. j) static. k) final. l) 非静态。

## 练习

- 6.2 创建一个名为 complex 的类, 完成复数的数学运算。编写一个驱动程序来测试该类。复



数具有下列形式:

$$\text{realPart} + \text{imaginaryPart} * i$$

其中  $i$  为:

$$\sqrt{-1}$$

使用浮点变量来表示类的私有数据。提供一个构造函数,使这个类的对象在创建时能够初始化。提供一个无参构造函数,在未提供初始值时使用默认值。提供 public 方法实现下列要求:

- a) 两个 Complex 数相加:实部和虚部分别相加。
  - b) 两个 Complex 数相减:左操作数的实部减去右操作数的实部,左操作数的虚部减去右操作数的虚部。
  - c) 按照(a, b)的形式打印 Complex 数,其中 a 是实部, b 是虚部。
- 6.3 创建一个名为 Rational 的类,完成分数的数学运算。编写一个驱动程序测试该类。用整型变量表示类的 private 实例变量——numerator 和 denominator。提供一个构造函数,使这个类的对象在创建时能够初始化。构造函数应当以约分形式存储分数(即分数 2/4 在对象中将存为分子为 1、分母为 2 的形式)。提供一个无参构造函数,在没有提供初始值时使用默认值。使用 public 方法完成下列功能:
- a) 两个 Rational 数相加,相加的结果应当以约分形式存储。
  - b) 两个 Rational 数相减,相减的结果应当以约分形式存储。
  - c) 两个 Rational 数相乘,相乘的结果应当以约分形式存储。
  - d) 两个 Rational 数相除,相除的结果应当以约分形式存储。
  - e) 用 a/b 的形式打印 Rational 数,其中 a 是分子, b 是分母。
  - f) 用浮点格式打印 Rational 数(考虑提供格式化的功能,使类的用户能够标明精确到小数点后的位数)。
- 6.4 修改图 6.5 中的 Time 类,增添一个 tick 方法,以 1 秒为基准增加存储在 Time 对象中的时间,还要提供 incrementMinute 方法和 incrementHour 方法来分别增加分钟值和小时值。Time 对象应当总是处于一个稳定的状态中。编写一个驱动程序,测试 tick 方法、incrementMinute 方法和 incrementHour 方法,以确保它们正确运行。测试下列各项:
- a) 递增到下一分钟。
  - b) 递增到下一小时。
  - c) 递增到下一天(即晚上 11:59:59 到凌晨 12:00:00)。
- 6.5 修改图 6.8 中的 MyDate 类,完成对实例变量 month、day 和 year 的初始化值的错误检查,还要提供一个 nextDay 方法以增加一天。MyDate 对象应当总是保持在一个稳定的状态中。编写一个驱动程序在一个循环中测试 nextDay 方法,并在每次循环时打印日期,以显示 nextDay 方法工作正确。测试以下项目:
- a) 递增到下个月。
  - b) 递增到下一年。
- 6.6 结合练习 6.4 中修改过的 Time 类和练习 6.5 中修改过的 MyDate 类,使它们成为一个 DateAndTime 类(在第 7 章中将讨论继承,以实现不修改现有的类定义就能迅速完成此任务)。修改 tick 方法,如果时间递增到下一天就调用 nextDay 方法。修改 toString 方法和

- PrintMilitary 方法以输出日期和时间。编写一个驱动程序来测试这个新类 DataAndTime, 尤其要测试时间增值而进入下一天的情况。
- 6.7 修改图 6.5 中程序的 set 方法, 如果试图向 Time 类对象的 hour、minute 和 second 实例变量设置非法值, 则应该返回一个相应的错误值。
- 6.8 创建一个 Rectangle 类。该类具有属性 length 和 width, 它们的默认值是 1。该类具有一些用来计算 perimeter 和 area 的方法, length 和 width 也有相应的 set 与 get 方法。set 方法验证 length 和 width 的值处于浮点数 0.0 和 20.0 之间。
- 6.9 创建一个比练习 6.8 更复杂的 Rectangle 类, 这个类只存储矩形四个角的笛卡儿坐标。构造函数调用一个 set 方法, 接受四组坐标值并验证它们都处于第一象限, 而且没有一个  $x$  或  $y$  坐标大于 200。另外 set 方法验证所提供的坐标是否定义了一个矩形, 其中两个坐标方向上较大的一个是长度。再设置一个谓词方法 square, 判断此矩形是否是一个正方形。
- 6.10 修改练习 6.9 中的 Rectangle 类, 加入一个 draw 方法, 在矩形所在第一象限的  $25 \times 25$  闭合框中显示此矩形。使用 Graphics 类的方法帮助输出 Rectangle。还可以加入一些方法, 在第一象限的指定区域内缩放、旋转并移动该矩形。
- 6.11 创建 HugeInteger 类, 用一个 40 元素的数组存储 40 位的整数, 提供 inputHugeInteger 方法、outputHugeInteger 方法、addHugeIntegers 方法和 subtractHugeIntegers 方法。为了比较, HugeInteger 对象应提供 isEqualto 方法、isNotEqualTo 方法、isGreaterThan 方法、isLessThan 方法、isGreaterThanOrEqualTo 方法和 isLessThanOrEqualTo 方法, 这些都是“谓词”方法, 只有关系成立时才返回 true, 而当两个 HugeInteger 间的关系不成立时, 则返回 false。提供一个谓词方法 isZero, 还可以再加上 MultiplyHugeIntegers 方法、divideHugeIntegers 方法和 modulusHugeIntegers 方法。
- 6.12 创建一个 TicTacToe 类, 编写一个程序来玩三连棋 (“井”字) 游戏。该类包括一个作为 private 数据的  $3 \times 3$  二维整数数组。构造函数应当将整个棋盘初始化为空。应允许两名选手参加。第一个选手走到某位置, 就在那个方格中放一个 1; 第二个选手走到某位置, 就在那个方格中放一个 2, 每次走动都必须是走向一个空格。在每次走动后判断游戏是否出现输赢, 或者游戏为平局。还可以修改该程序, 使计算机自动替代一名选手走棋, 以及允许选手自己确定是先走还是后走。如果读者有兴趣, 还可以在一个  $4 \times 4 \times 4$  的棋盘上开发一个玩三维三连棋游戏的程序 (注意: 这个挑战性的题目可能要数星期才能完成)。
- 6.13 解释 Java 中软件包访问的概念, 解释本书中所指的软件包访问的副作用。
- 6.14 如果为构造函数或终止函数定义了返回类型, 就算是 void, 那么将发生什么情况?
- 6.15 创建一个 MyDate 类, 实现下列功能:
- 用多种格式输出日期, 例如:  

```
MM / DD / YY  
June 14, 1992  
DDD YYYY
```
  - 用重载的构造函数创建 MyDate, 并使用 a) 中的日期格式进行初始化。
- 6.16 创建 SavingsAccount 类。用一个静态类变量来存储每个账户的 annualInterestRate。每个类的对象都包括一个 private 实例变量 SavingsBalance, 用来指明账户当前的储蓄额。提供 calculateMonthlyInterest 方法, 用 annualInterestRate 除以 12 再乘以 balance 来计算月利, 并

将这个利率加到 `savingsBalance` 中。提供一个 `static` 方法 `modifyInterestRate`, 用来为 `annualInterestRate` 设置新值。编写一个驱动程序来测试 `SavingsAccount` 类。实例化两个不同的 `SavingsAccount` 对象 `saver1` 和 `saver2`, 各自拥有的余额为 \$2 000.00 和 \$3 000.00。将 `annualInterestRate` 设置为 4%, 然后为每个储户计算月利并打印出新的余额。接着将 `annualInterestRate` 设置为 5%, 计算下个月的利率并为每个储户打印出新的余额。

- 6.17 创建 `IntegerSet` 类, 类中的每个对象包括 0 到 100 的整数, 每个整数集合由一个值为 0 和 1 的数组表示。如果整数  $i$  在集合内, 则数组元素  $a[i]$  为 1。如果整数  $j$  不在集合内则数组元素  $a[j]$  为 0。无参构造函数将一个集合初始化为所谓的“空集”, 即数组中的全部元素都是 0。

程序中应提供下列方法:

`UnionOfIntegerSets` 方法创建第三个集合, 它是集合论中两个现有集合的并集 (即如果有元素在某个或两个集合中为 1, 那么在第三个集合中将该元素置为 1, 否则置为 0)。

`intersectionOfIntegerSets` 方法创建第三个集合, 它是集合论中两个已有集合的交集 (即元素在某个集合中为 0 或在两个集合中都为 0, 则在第三个集合中将该元素置为 0, 否则置为 1)。

`insertElement` 方法向集合中插入一个新整数  $k$  (即将  $a[k]$  设置为 1)。

`deleteElement` 方法删除整数  $m$  (即将  $a[m]$  设置为 0)。

`setPrint` 方法打印用空格隔开的一系列数组。只打印出集合中存在的元素, 如果是空集就打印 ---。

`isEqualTo` 方法判断两个集合是否相等。

编写一个程序测试 `IntegerSet` 类, 实例化几个 `IntegerSet` 对象, 测试所有方法, 看其是否工作正常。

- 6.18 利用从午夜开始的秒数来表示图 6.1 中 `Time` 类的时间, 这样要比使用 3 个整数 `hour`、`minute` 和 `second` 更合理一些。客户可以使用同样的 `public` 方法并获得同样的结果。修改图 6.1 中的 `Time` 类, 用从午夜开始的秒数来实现 `Time` 类, 而且类的客户应该看不出有什么的改动。

## 第7章 面向对象的编程

### 教学目标

- 通过继承现有的类而创建新类
- 理解继承和软件可重用性
- 理解超类和子类
- 理解多态特性如何使系统变成可扩展的和可维护的
- 理解抽象类和具体类的差别
- 创建抽象类

### 7.1 简介

在本章中，我们要讨论面向对象的编程（object-oriented programming, OOP）及其关键的组件技术——继承（inheritance）和多态（polymorphism）。继承是软件重用的一种形式，这种技术通过包含现有类的属性和行为来创建新类，并通过新类的功能来丰富现有类。软件重用节省了程序开发时间，鼓励程序员重用那些经过证明和调试的高质量软件，从而减少系统投入运行后所带来的问题。多态使得我们可以按照通用的风格来编写程序，以此处理一大批现有的以及还未说明的相关类，并且很容易向系统添加新的功能。继承和多态都是处理软件复杂性的有效技术。

当创建一个新类时，程序员可以指明新类将继承以前定义为超类（superclass）的实例变量和实例方法，而不是写出所有的新实例变量和新实例方法。新的类称为子类（subclass），每个子类也可成为其他类的超类。

子类的直接超类（direct superclass）是子类显式继承的超类，间接超类是在类层次中从两层或更多层次之外继承的超类。

单继承（single inheritance）是指一个类派生于一个超类。Java 不支持多继承（C++ 支持多继承），但它支持接口的概念。接口帮助 Java 获得了多继承的优点，而且并没有带来一些相关的问题。关于接口的详细内容请参见第 13 章，我们将考虑两个通用原则，并讨论带有线程的 Runnable 接口的案例。

一般情况下，子类会添加自己所用的变量和方法，因此子类一般大于它的超类。子类比其超类更加专门化，它只代表一组更小的对象组。通过单继承，子类最初同超类一样。继承的作用在于除了能替换从超类继承的特性之外，还可以在子类中额外定义新的内容。

子类的每个对象也是其超类的一个对象。但是，反过来超类的对象并不是其子类的对象。可以利用这种“子类对象是超类对象”的关系来完成一些功能强大的操作，例如，我们可以将与同一个公共超类有关的各种不同对象，通过继承而连接成一个超类对象的链表。这就允许使用一种通用方式来处理各种不同的对象。正如我们将在本章和下一章中看到的，这是推动面向对象编程的一个关键动力。

我们将在本章中引入一个新形式的成员访问控制，称为 protected（受保护的）访问，子类的方法以及在同一个软件包中的其他类的方法可以访问 protected 超类成员。

根据构建软件系统的一些经验,程序代码中相当大的一部分用于处理密切相关的特殊情况。在这类系统中难以见到“全景”,因为设计人员和程序员都致力于那些特殊情况。面向对象的编程提供了几种“透过树木看森林”的方法,即一种称为抽象(abstraction)的处理过程。

如果一个过程化的程序含有许多密切相关的特殊情况,那么一般使用switch语句来区分这些特殊情况,并分别为每个情况提供逻辑处理。下面,我们将介绍如何利用继承和多态并通过较简单的逻辑来替换这种switch逻辑。

我们应当区分“is a”(是)关系和“has a”(有)关系。“is a”关系即继承,在“is a”关系中,可以将子类类型的对象视为超类类型的对象。“has a”则是复合(正如我们在第6章中讨论的),在“has a”关系中,类对象拥有一个或多个其他类的对象作为成员。

一个子类的方法也许需要访问其超类中特定的实例变量和方法。子类不能直接访问其超类的private(私有)成员,这是Java软件工程的关键之处。如果一个子类可以访问超类的private成员,则会破坏隐藏在超类中的信息。

#### 软件工程观点 7.1

子类不能直接访问其超类的private成员。

#### 测试与调试提示 7.1

隐藏private成员对于测试、调试和正确地修改系统有着巨大的帮助。如果一个子类可以访问其超类的private成员,那么由子类派生出的类也可能访问该数据。这将不断产生对private数据的访问,同时信息隐藏的优势也将消失在类的层次中。

然而,子类能够访问其超类的public和protected成员。超类中不应由子类通过继承而访问的成员将声明为private,只有通过超类中提供的public、protected和软件包访问方法以及子类接口,才能改变超类中private成员的状态。

继承中存在一个问题,即一个子类有时继承了它不需要甚至是不应拥有的方法。当超类的成员不适合于它的子类时,可以通过一种适当的实现在子类中重写(覆盖)那些成员。

也许最令人振奋的还是能从丰富的类库(class library)中继承新类。各种组织开发自己的类库,同时也利用任何可获得的其他类库。最终,大多数软件可以由标准化的可重用组件构成,就像现在的硬件一样,这将有助于将来开发功能更强大的软件。

## 7.2 超类和子类

一个类的对象常常也“是”另一个类的对象,如同矩形也是一个四边形(正方形、平行四边形和梯形都是四边形),因此,可以认为Rectangle类继承自Quadrilateral类。在这种情况下,Quadrilateral类是一个超类,而Rectangle类是一个子类。一个矩形是一个特殊类型的四边形,但一个四边形并不是一个矩形(四边形也可能是一个平行四边形)。图7.1给出了几个简单的继承例子。

因为在正常情况下,继承产生出比超类含有更多特性的子类,所以术语“超类”和“子类”可能会产生混淆。我们还可以使用另一种方式来理解这些术语,因为每个子类对象是其超类的对象,并且一个超类可能拥有许多子类,所以由超类所表示的对象集合一般要比由该超类的任何子类所表示的对象集合都要大。

继承关系组成树状的层次结构,一个超类与其子类构成了层次关系。一个类当然可以独立存在,但当在继承机制中使用一个类时,该类要么成为向其他类提供属性和行为的超类,要么成为继承属性和行为的子类。

| 超类 | 子类                     |
|----|------------------------|
| 学生 | 研究生<br>本科生             |
| 形状 | 圆形<br>三角形<br>矩形        |
| 信贷 | 汽车信贷<br>家庭发展信贷<br>抵押信贷 |
| 雇员 | 教员<br>职员               |
| 账目 | 支票账目<br>储蓄账目           |

图 7.1 一些继承的简单例子

让我们开发一个简单的继承层次。一个典型的大学社区中有数千名成员，这些人包括雇员、学生和校友，雇员要么是教员要么是职员，教员包括管理者（如系主任）和教学人员，这样就产生了如图 7.2 所示的继承层次。注意，继承层次可以包含许多其他的类。例如，学生可以是研究生或本科生，本科生可以是一年级、二年级、三年级或四年级的学生等。

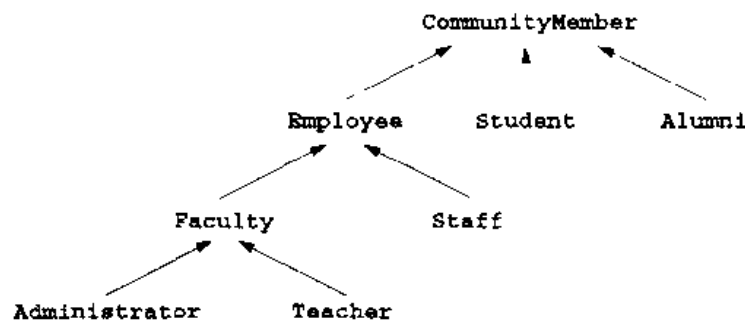


图 7.2 一个大学社区成员的继承层次

另一个基本的继承层次是图 7.3 所示的 Shape 层次。在现实世界中有着丰富的继承例子，但是学生们不习惯这样划分现实世界，于是调整他们的思维方式。实际上，生物系的学生有些层次方面的经验。我们在生物学领域中的研究对象都可以划分到一个层次中，最开始是生物，生物包括植物和动物等。

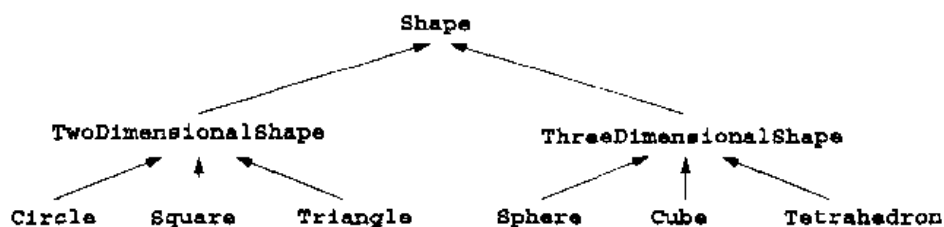


图 7.3 Shape 类的层次的一部分

为表明 CommissionWorker 类派生自（或继承自）Employee 类，CommissionWorker 类在 Java 中可以定义为：

```
class CommissionWorker extends Employee { ... }
```

继承中只有超类的`private`成员不能由其子类访问, 所有其他的超类成员均成为子类的成员, 我们可以将超类对象和子类对象看成是相似的, 其共同之处表现在超类的属性和行为中, 派生自一个公有超类的所有类对象都可认为是该超类的对象。

下面给出了一些例子, 我们能够利用继承关系使这些例子易于编写, 但在C这样的非面向对象的语言中则做不到这一点。

### 7.3 protected 成员

超类的`public`成员可以由程序中的所有方法访问, 而`private`成员则只能由超类的方法访问。

`protected`访问是介于`public`访问和`private`访问之间的一种中间保护形式, 超类的`protected`成员可以由该超类的方法访问, 也可以由其子类的方法和同一个软件包内其他类的方法访问。

正常情况下, 子类的方法使用成员名就可以访问其超类的`public`和`protected`成员。当子类的方法覆盖了超类的方法时, 子类可以通过在超类名前加上`super`、后跟句点运算符(`.`)的方式来访问超类的方法。

### 7.4 超类对象和子类对象之间的关系

子类的对象可以看成是其超类的一个对象, 这使我们可以完成一些有趣的操作。例如, 尽管从某个特定超类派生出的各种类的对象可能各不相同, 但我们却能够创建这些对象的数组——只要将它们视为超类的对象即可。但反过来却不正确: 超类的对象不能自动成为子类对象。例如, 一种形状并不总是圆形。

但是, 可以显式地将一个超类引用转换成一个子类引用。这仅当超类引用实际上是引用一个子类对象时才可以实现, 否则Java将抛出一个`ClassCastException`异常。

#### 常见编程错误 7.1

将超类对象(未经转换)赋值给子类引用是一个语法错误

#### 软件工程视点 7.2

如果已经将一个对象赋值给其超类的引用, 那么将该对象转换为其自身类型是可行的。实际上, 为了向该对象发送超类中所没有的消息, 也必须这样做。

在图7.4中给出了我们的第一个例子。程序的第1行~第26行显示了`Point`类和`Point`方法的定义, 第27行~第63行显示了`Circle`类和`Circle`方法的定义, `Circle`类继承自`Point`类。第64行~第101行显示了一个驱动程序, 演示了如何将子类引用赋给超类引用, 并将超类引用转换成子类引用。

让我们先在第1行~第26行中查看`Point`的类定义。`Point`的`public`接口包括`Point`方法、`setPoint`方法、`getX`方法、`getY`方法和`toString`方法。将`Point`的实例变量`x`和`y`声明为`protected`, 这样可以防止`Point`对象的客户访问这些数据(除非它们在一个软件包中), 但是却使从`Point`派生的类可以直接访问继承的实例变量。如果将数据定义为`private`, 那么就必须使用`Point`的非`private`方法访问这些数据, 甚至子类也是如此。

Circle 类（第 27 行 ~ 第 63 行）继承自 Point 类，这在类定义的头一行中进行了说明：

```
public class Circle extends Point ; //inherits from Point
```

关键字 `extends` 在类定义中指明继承。Point 类的所有（非 private）成员都通过继承而成为 Circle 的成员。因此，Circle 的 public 接口包括 Point 的 public 方法，也包括两个重载的 Circle 构造函数，以及 Circle 的方法 `setRadius`、`getRadius`、`area` 和 `toString`。

Circle 构造函数（第 33 行 ~ 第 38 行和第 40 行 ~ 第 45 行）必须激活 Point 的构造函数来初始化 Circle 对象的超类部分，每个构造函数（第 36 行和第 43 行）的头一行通过对 `super` 的引用来激活 Point 的构造函数，无参构造函数通过传递 0 来初始化超类的成员 `x` 和 `y`，另一个构造函数将值 `a` 和 `b` 传递给 Point 构造函数，以初始化超类的成员 `x` 和 `y`。

```

1 // Fig. 7.4: Point.java
2 // Definition of class Point
3
4 public class Point {
5     protected double x, y; // coordinates of the Point
6
7     // constructor
8     public Point( double a, double b ) { setPoint( a, b ); }
9
10    // Set x and y coordinates of Point
11    public void setPoint( double a, double b )
12    {
13        x = a;
14        y = b;
15    }
16
17    // get x coordinate
18    public double getX() { return x; }
19
20    // get y coordinate
21    public double getY() { return y; }
22
23    // convert the point into a String representation
24    public String toString()
25    { return "[" + x + ", " + y + "]; }
26 }
27 // Fig. 7.4: Circle.java
28 // Definition of class Circle
29
30 public class Circle extends Point { // inherits from Point
31     protected double radius;
32
33     // no-argument constructor
34     public Circle()
35     {
36         super( 0, 0 ); // call the base class constructor
37         setRadius( 0 );
38     }
39
40     // Constructor
41     public Circle( double r, double a, double b )

```

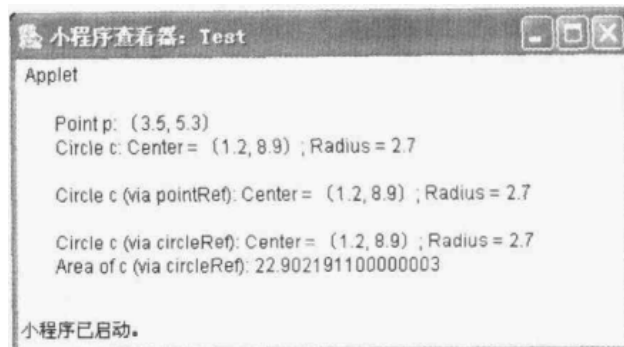


```
42     {
43         super( a, b ); // call the base class constructor
44         setRadius( r );
45     }
46
47     // Set radius of Circle
48     public void setRadius( double r )
49     { radius = ( r >= 0.0 ? r : 0.0 ); }
50
51     // Get radius of Circle
52     public double getRadius() { return radius; }
53
54     // Calculate area of Circle
55     public double area() { return 3.14159 * radius * radius; }
56
57     // convert the Circle to a String
58     public String toString()
59     {
60         return "Center = " + "[" + x + ", " + y + "]" +
61             "; Radius = " + radius;
62     }
63 }
64 // Fig. 7.4: Test.java
65 // Casting superclass references to subclass references
66 import java.awt.Graphics;
67 import java.applet.Applet;
68
69 public class Test extends Applet {
70     private Point pointRef, p;
71     private Circle circleRef, c;
72
73     public void init()
74     {
75         p = new Point( 3.5, 5.3 );
76         c = new Circle( 2.7, 1.2, 8.9 );
77     }
78
79     public void paint( Graphics g )
80     {
81         g.drawString( "Point p: " + p.toString(), 25, 25 );
82         g.drawString( "Circle c: " + c.toString(), 25, 40 );
83
84         // Attempt to treat a Circle as a Point
85         pointRef = c; // assign Circle to pointRef
86         g.drawString( "Circle c (via pointRef): " +
87             pointRef.toString(), 25, 70 );
88
89         // Treat a Circle as a Circle (with some casting)
90         pointRef = c; // assign Circle to pointRef
91         circleRef = (Circle) pointRef; // cast super to sub
92         g.drawString( "Circle c (via circleRef): " +
93             circleRef.toString(), 25, 100);
94         g.drawString( "Area of c (via circleRef): " +
95             circleRef.area(), 25, 115 );
96
97         // Attempt to refer to Point object
```

```

98          // with Circle reference
99          circleRef = (Circle) p;
100      }
101  }

```



#### 输出结果:

```

Exception in thread "AWT-Callback-Win32"
    java.lang.ClassCastException: Point
        at Test.paint(Test.java:36)
        at sun.awt.win32.MComponentPeer.paint
            (MComponentPeer.java:147)
        at sun.awt.win32.MComponentpeer.handleExpose
            (MComponentPeer.java:268)
        at sun.awt.win32.MToolkit.run(MToolkit.java:57)

```

图 7.4 将子类引用赋给超类引用

如果 Circle 构造函数没有显式地激活 Point 构造函数, 那么 Point 构造函数利用无参数方式激活。因为 Point 类没有提供无参构造函数, 所以编译器将产生一个错误。

通过使用同样的特征 (signature), 子类可以重定义超类方法, 即重写超类方法。当该方法的名字在子类中出现时, 将自动使用子类的版本。super 引用后跟句点运算符可用于从子类中访问该方法的超类版本。

注意, Circle 类重写了 Point 类的 toString 方法 (第 57 行 ~ 第 62 行)。Circle 方法 toString 直接访问了 protected 实例变量 x 和 y, 这些变量是从 Point 类继承而来的。

#### 软件工程视点 7.3

超类方法在子类中的重定义不需要有与超类方法一样的特征, 这样的重定义不是方法重写, 只不过是一个方法重载的例子。

#### 常见编程错误 7.3

如果超类中的方法与其子类中的方法有着同样的特征但返回类型不同, 则是语法错误。

applet (第 64 行 ~ 第 101 行) 实例化了 Point 对象 p 和 Circle 对象 c (第 75 行 ~ 第 76 行), 然后输出这些对象以显示它们经过了正确的实例化 (第 81 行 ~ 第 82 行)。

然后, 将一个子类对象 Circle c 赋给 pointRef (第 85 行)。在 Java 中将子类引用赋给超类引用是可以接受的 (因为是继承的 “is a” 关系), 但反过来却是危险的, 这一点我们将会在后面看到。有趣的是, 当 x 向这个 pointRef 发送 toString 消息时, Java 知道该对象实际上是一个 Circle, 于是它选择 Circle 的 toString 方法而不是使用 Point 的 toString 方法, 这可能与我们的想像的有所不同。这是一

个多态和动态绑定的例子，它们是我们在本章要深入研究的概念。

在第90行我们将 Circle 引用 c 赋给 pointRef（第二次赋值，这类赋值在 Java 中是允许的）。在第91行上我们将 pointRef（引用一个 Circle）转换成 CircleRef（如果 pointRef 引用了一个 Point，那么这种转换是危险的），然后利用 CircleRef 来打印与 Circle 的 CircleRef 有关的各种结果。

接着，我们试着在第99行进行一次危险的转换，将 Point p 转换成 CircleRef。在运行时遇到这种语句时，Java 会认为 p 实际上引用一个 Point；而转换成 Circle 则是危险的，这将抛出一个 ClassCastException 异常。因为在此处没有给出如何处理这种异常，所以程序将会终止。我们将在第12章深入讨论关于异常处理的详细内容。当发生 ClassCastException 异常时，第二个输出窗口中将显示错误消息，这种错误消息一般包括文件名和错误发生的行号，以使用户能够找到程序中的相应行，从而去除错误。[注意，所指出的第一行错误 test.java:36 不同于本书中的 test.java 的行号。这是因为本书中的例子为了便于讨论，将程序中的各个文件按顺序编行号。如果用户打开了 test.java，将会发现错误的确实发生在第36行上。]

## 7.5 在子类中使用构造函数和终止函数

当实例化子类的一个对象时，应当调用超类的构造函数，以完成子类对象中超类实例变量的初始化工作。对超类构造函数（通过 super 引用）的一个显式调用，可以作为子类构造函数的第一条语句。否则子类构造函数将隐式地调用超类的默认构造函数（或无参构造函数）。

### 常见编程错误 7.4

如果子类构造函数激活了超类的默认构造函数（或无参构造函数），并且超类没有默认构造函数（或无参构造函数），将会发生语法错误。

### 常见编程错误 7.5

如果子类对超类构造函数的 super 调用不在子类构造函数的第一条语句上，则会产生语法错误。

### 软件工程视点 7.4

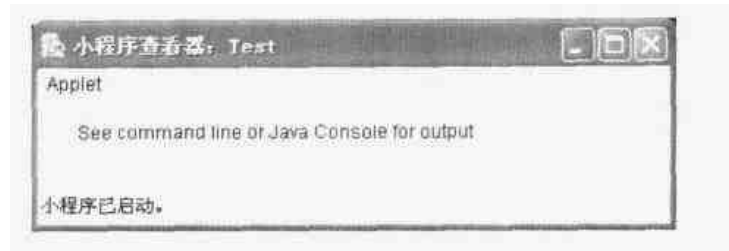
当创建子类的一个对象时，首先执行超类构造函数，然后执行子类构造函数。

图 7.5 显示了超类和子类构造函数及终止函数的调用顺序。Point 类（第1行~第37行）包含了一个构造函数、一个终止函数，以及其他的 public 方法和 protected 实例变量 x 和 y。构造函数和终止函数分别打印它们的执行情况，然后显示它们激活的 Point。Circle 类（第38行~第86行）派生自 Point 并且包括两个构造函数和一个终止函数，以及其他的 public 方法和 protected 实例变量 radius。构造函数和终止函数分别打印它们的执行情况，然后显示它们激活的 Circle。Circle 构造函数也通过 super 和传递值 a 和 b 来激活 Point 构造函数，以便初始化超类的实例变量。注意，Circle 方法 toString 通过 super（第83行）来激活 Point 的 toString。

```
1 // Fig. 7.5: Point.java
2 // Definition of class Point
3 public class Point {
4     protected double x, y; // coordinates of the Point
5
6     // constructor
7     public Point( double a, double b )
8     {
```

```
9         setPoint( a, b );
10        System.out.println( "Point constructor: " +
11                               toString() );
12    }
13
14    // finalizer
15    public void finalize()
16    {
17        System.out.println( "Point finalizer: " +
18                               toString() );
19    }
20
21    // Set x and y coordinates of Point
22    public void setPoint( double a, double b )
23    {
24        x = a;
25        y = b;
26    }
27
28    // get x coordinate
29    public double getX() { return x; }
30
31    // get y coordinate
32    public double getY() { return y; }
33
34    // convert the point into a String representation
35    public String toString()
36    { return "[" + x + ", " + y + "]" ; }
37 }
38 // Fig. 7.5: Circle.java
39 // Definition of class Circle
40
41 public class Circle extends Point { // Inherits from Point
42     protected double radius;
43
44     // no-argument constructor
45     public Circle()
46     {
47         super( 0, 0 ); // call the base class constructor
48         setRadius( 0 );
49         System.out.println( "Circle constructor: " +
50                               toString() );
51     }
52
53     // Constructor
54     public Circle( double r, double a, double b )
55     {
56         super( a, b ); // call the base class constructor
57         setRadius( r );
58         System.out.println( "Circle constructor: " +
59                               toString() );
60     }
61
62     // finalizer
63     public void finalize()
```

```
64     {
65         System.out.println( "Circle finalizer: " +
66                             toString() );
67     }
68
69     // Set radius of Circle
70     public void setRadius( double r )
71     { radius = ( r >= 0 ? r : 0 ); }
72
73     // Get radius of Circle
74     public double getRadius() { return radius; }
75
76     // Calculate area of Circle
77     public double area()
78     { return 3.14159 * radius * radius; }
79
80     // convert the Circle to a String
81     public String toString()
82     {
83         return "Center = " + super.toString() +
84             "; Radius = " + radius;
85     }
86 }
87 // Fig. 7.5: Test.java
88 // Demonstrate when superclass and subclass
89 // constructors and finalizers are called.
90 import java.awt.Graphics;
91 import java.applet.Applet;
92
93 public class Test extends Applet {
94     private Circle circle1, circle2;
95
96     public void init()
97     {
98         circle1 = new Circle( 4.5, 7.2, 2.9 );
99         circle2 = new Circle( 10, 5, 5 );
100     }
101
102     public void start()
103     {
104         circle2 = null; // Circle can now be garbage collected
105         circle1 = null; // Circle can now be garbage collected
106
107         System.gc();    // call the garbage collector
108     }
109
110     public void paint( Graphics g )
111     {
112         g.drawString(
113             "See command line or Java Console for output",
114             25, 25 );
115     }
116 }
```



输出结果:

```
Point constructor: Center = [7.2, 2.9]; Radius = 0
Circle constructor: Center = [7.2, 2.9]; Radius = 4.5
Point constructor: Center = [5.5]; Radius = 0
Circle constructor: Center = [5, 5]; Radius = 10
Circle finalizer: Center = [7.2, 2.9]; Radius = 4.5
Circle finalizer: Center = [5, 5]; Radius = 10
```

图 7.5 构造函数和终止函数的调用顺序

#### 常见编程错误 7.6

当子类覆盖超类的方法时,使用子类版本调用超类版本,并再进行一些附加的工作,这是很常见的。不要使用 `super` 引用来引用超类的方法,这样做会导致无限递归,因为子类方法实际上是在调用自己。

#### 常见编程错误 7.7

通过级联 `super` 引用 (如 `super.super.x`) 来引用层次中的一个成员 (方法或变量) 是语法错误。

第 87 行 ~ 第 116 行给出了一个测试这个 `Point/Circle` 继承层次的 applet。程序首先初始化 `Circle` 对象 `circle1`, 这将激活 `Circle` 构造函数 (第 53 行 ~ 第 60 行), 后者又激活了 (第 56 行) `Point` 构造函数, 并使用从 `Circle` 构造函数传递来的值来完成输出。接着实例化 `Circle` 对象 `circle2`, 再次运行 `Point` 和 `Circle` 构造函数。注意, `Point` 构造函数体在 `Circle` 构造函数体之前执行, 表示对象的构造是“由内及外”。

在 `start` 方法中, 我们先将 `circle2` 设置为 `null`, 然后再将 `circle1` 设置为 `null`。由于这些对象都不再需要, `Java` 对 `circle2` 和 `circle1` 所占的内存进行标记, 表示它们可作为无用单元而回收。Java 保证在无用单元回收器回收这些对象的空间之前, 调用每个对象的终止函数。无用单元回收器是一个低优先级的线程, 无论何时处理器时间都可以自动运行, 在本例中使用第 107 行上的 `System.gc()` 来强制无用单元回收器运行。注意, `Java` 并不保证对象作为无用单元进行回收时的顺序, 也不确定终止函数的执行顺序。

## 7.6 从子类对象到超类对象的隐式转换

尽管一个子类对象也是一个超类对象, 但子类类型和超类类型却不相同, 子类对象可认为是超类对象, 因为子类都有对应每个超类成员的成员。请记住, 子类一般拥有比超类更多的成员。而反方向的赋值则是不允许的, 因为将超类对象赋给子类引用会导致另外的子类成员没有定义。

对子类对象的引用可以隐式地转换成对超类对象的引用, 因为子类对象 (通过继承关系) 也是超类对象。

这里存在 4 种可能方式, 可以利用超类对象和子类对象来混合以及匹配超类引用和子类引用:

1. 使用超类引用来引用超类对象。

2. 使用子类引用来引用子类对象。
3. 使用超类引用来引用子类对象是安全的, 因为子类对象也是其超类的对象。这种代码只能引用超类成员。如果这种代码通过超类引用来引用只在子类中才有的成员, 那么编译器将报告语法错误。
4. 使用子类引用来引用超类对象是语法错误。子类引用必须先转换成一个超类引用。

将子类对象看成超类对象也许是方便的, 可以通过超类引用操作这些对象来实现这一点, 但这里却有一个问题。例如, 在一个工资支付系统中, 我们希望能够遍历雇员数组并为每个人计算出每周薪水。如果使用超类引用, 则程序只能调用超类支付计算例程 (如果在超类中确有这样一个例程), 我们需要为每个对象激活合适的支付计算例程, 无论它是一个超类对象还是一个子类对象, 只通过超类引用就可实现这一点。在我们考虑多态和动态绑定时, 还要在本章稍后部分讨论上述内容。

## 7.7 使用继承的软件工程

我们能够使用继承来定制现有的软件。当使用继承从现有的类创建一个新类时, 新类就继承了一个现有类的属性和行为, 接下来我们能够添加属性和行为, 或者重写超类的行为来定制该类, 以满足不同的需要。

设计人员和实现者在大型工业软件项目中所面对的问题是难以想像的, 有经验的开发人员不约而同地指出, 改进软件开发过程的关键是鼓励软件重用。从宏观上来看, 面向对象编程 (特别是 Java) 确实做到了这一点。

正是通过继承基本和可用的类库, 才带来了软件重用的最大效益。随着我们对 Java 的兴趣的日益增加, 也将会逐步关注 Java 类库。就像独立软件商开发的打包 (shrink-wrapped) 软件随着个人计算机的到来而呈爆炸式的增长, Java 类库的创建和销售也将惊人地增长。应用设计人员将会使用这些库来建立他们自己的应用程序。我们将面临为各种应用领域开发 Java 类库, 这是巨大的全球性任务。

### 软件工程视点 7.5

创建子类并不影响其超类的源代码或 Java 字节码, 继承保护了超类的完整性。

超类定义了共性。从超类派生出的所有类继承了该超类的功能。在面向对象的设计过程中, 设计人员要寻找一组类的共性, 提取出它们并组成需要的超类。然后再定制子类, 从而超出从超类继承的性能。

### 软件工程视点 7.6

如同非面向对象系统的设计人员应当避免不必要的函数激增一样, 面向对象系统的设计人员应当避免不必要的类激增。激增的类带来了管理问题, 同时也会阻碍软件工程, 因为对于一个潜在类的重用者, 难以在巨大的集合中定位这个类。折衷观点就是创建较少的类, 每个类都提供基本的附加功能, 但是这些类对特定领域的重用者而言还是过于丰富了。

### 性能提示 7.1

如果通过继承产生的类大于所需要的大小, 这将浪费存储器和处理资源。应该继承与所需类“最密切”的类。

注意, 阅读一组子类的声明可能会引起混淆, 因为继承的成员并未显示出来, 但是这些成员是不会在子类中显示的。类似的问题也存在于子类的文档中。

**软件工程视点 7.7**

在一个面向对象的系统中，类通常是密切相关的，“抽出”共同的属性和行为并将它们放在超类中，然后利用继承来组成子类，可以不用重复定义共同的属性和行为。

**软件工程视点 7.8**

只要没有改动超类的公有接口，对超类的修改就不要要求对子类进行相应的修改。

## 7.8 复合与继承

我们已经讨论了由继承实现的“is a”关系，同时也讨论了“has a”关系（参见前面章节的例子）。在后面的一种关系中，一个类可能将其他类的对象作为自己的成员，这种关系通过复合（composition）现有的类来创建新类。例如，给定类 Employee、BirthDate 及 TelephoneNumber，指出 Employee 类是 BirthDay 类或者 Employee 类是 TelephoneNumber 类都是不合适的。但是指出 Employee 类含有 BirthDay 类或者 Employee 类含有 TelephoneNumber 类却是合理的。

## 7.9 案例分析：点、圆、圆柱体

现在给出一个基本的继承例子，我们考虑一个点、圆和圆柱体的层次结构。首先开发并使用 Point 类（如图 7.6 所示）。接着提供一个例子，其中从 Point 类派生出 Circle 类（如图 7.7 所示）。最后再提供一个例子，从 Circle 类派生出 Cylinder 类（如图 7.8 所示）。

```
1 // Fig. 7.6: Point.java
2 // Definition of class Point
3
4 public class Point {
5     protected double x, y; // coordinates of the Point
6
7     // constructor
8     public Point( double a, double b ) { setPoint( a, b ); }
9
10    // Set x and y coordinates of Point
11    public void setPoint( double a, double b )
12    {
13        x = a;
14        y = b;
15    }
16
17    // get x coordinate
18    public double getX() { return x; }
19
20    // get y coordinate
21    public double getY() { return y; }
22
23    // conver the point into a String representation
24    public String toString()
25    { return "[" + x + ", " + y + "]"; }
26 }
27 // Fig. 7.6: Test.java
28 // Applet to test class Point
```



```

29  import java.awt.Graphics;
30  import java.applet.Applet;
31
32  public class Test extends Applet {
33      private Point p;
34
35      public void init()
36      {
37          p = new Point( 7.2, 11.5 );
38      }
39
40      public void paint( Graphics g )
41      {
42          g.drawString( "X coordinate is " + p.getX(), 25, 25 );
43          g.drawString( "Y coordinate is " + p.getY(), 25, 40 );
44
45          p.setPoint( 10, 10 );
46          g.drawString( "The new location of p is " +
47                      p.toString(), 25, 70 );
48      }
49  }

```

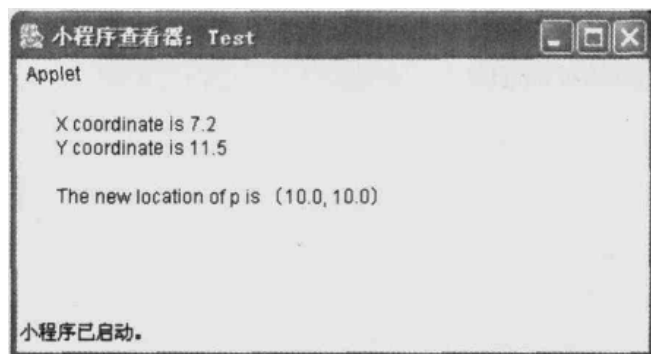


图 7.6 测试 Point 类

```

1  // Fig. 7.7: Circle.java
2  // Definition of class Circle
3
4  public class Circle extends Point { // inherits from Point
5      protected double radius;
6
7      // no-argument constructor
8      public Circle()
9      {
10         super( 0, 0 ); // call the base class constructor
11         setRadius( 0 );
12     }
13
14     // Constructor
15     public Circle( double r, double a, double b )
16     {
17         super( a, b ); // call the base class constructor
18         setRadius( r );
19     }
20
21     // Set radius of Circle

```

```

22     public void setRadius( double r )
23     { radius = ( r >= 0.0 ? r : 0.0 ); }
24
25     // Get radius of Circle
26     public double getRadius() { return radius; }
27
28     // Calculate area of Circle
29     public double area()
30     { return 3.14159 * radius * radius; }
31
32     // convert the Circle to a String
33     public String toString()
34     {
35         return "Center = " + super.toString() +
36             "; Radius = " + radius;
37     }
38 }
39 // Fig. 7.7: Test.java
40 // Applet to test class Circle
41 import java.awt.Graphics;
42 import java.applet.Applet;
43
44 public class Test extends Applet {
45     private Circle c;
46
47     public void init()
48     {
49         c = new Circle( 2.5, 3.7, 4.3 );
50     }
51
52     public void paint( Graphics g )
53     {
54         g.drawString( "X coordinate is " + c.getX(), 25, 25 );
55         g.drawString( "Y coordinate is " + c.getY(), 25, 40 );
56         g.drawString( "Radius is " + c.getRadius(), 25, 55 );
57
58         c.setRadius( 4.25 );
59         c.setPoint( 2, 2 );
60         g.drawString( "The new location and radius of c are ",
61             25, 85 );
62         g.drawString( c.toString(), 40, 100 );
63         g.drawString( "Area is " + c.area(), 25, 115 );
64     }
65 }

```

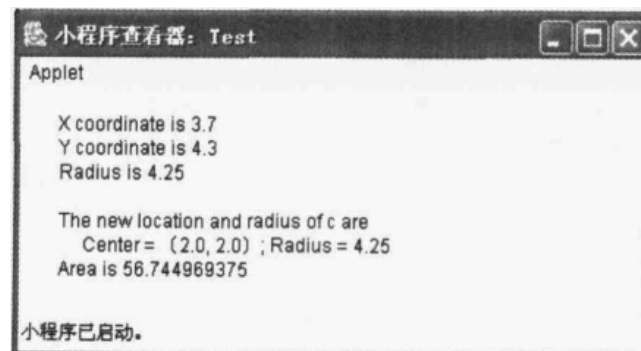


图 7.7 测试 Circle 类

```
1 // Fig. 7.8: Cylinder.java
2 // Definition of class Cy_linder
3 public class Cylinder extends Circle {
4     protected double height; // height of Cylinder
5
6     // Cylinder constructor calls Circle constructor
7     public Cylinder( double h, double r, double a, double b )
8     {
9         super( r, a, b );
10        setHeight( h );
11    }
12
13    // Set height of Cylinder
14    public void setHeight( double h )
15        { height = ( h >= 0 ? h : 0 ); }
16
17    // Get height of Cylinder
18    public double getHeight() { return height; }
19
20    // Calculate area of Cylinder (i.e., surface area)
21    public double area()
22    {
23        return 2 * super.area() +
24            2 * 3.14159 * radius * height;
25    }
26
27    // Calculate volume of Cylinder
28    public double volume() { return super.area() * height; }
29
30    // Convert the Cylinder to a String
31    public String toString()
32    {
33        return super.toString() + "; Height = " + height;
34    }
35 }
36 // Fig. 7.8: Test.java
37 // Applet to test class Cylinder
38 import java.awt.Graphics;
39 import java.applet.Applet;
40
41 public class Test extends Applet {
42     private Cylinder c;
43
44     public void init()
45     {
46         c = new Cylinder( 5.7, 2.5, 1.2, 2.3 );
47     }
48
49     public void paint( Graphics g )
50     {
51         g.drawString( "X coordinate is " + c.getX(), 25, 25 );
52         g.drawString( "Y coordinate is " + c.getY(), 25, 40 );
53         g.drawString( "Radius is " + c.getRadius(), 25, 55 );
54         g.drawString( "Height is " + c.getHeight(), 25, 70 );
55
56         c.setHeight( 10 );
```

```

57         c.setRadius( 4.25 );
58         c.setPoint( 2, 2 );
59
60         g.drawString( "The new location, radius and height" +
61                       " of c are ", 25, 100 );
62         g.drawString( c.toString(), 40, 115 );
63         g.drawString( "Area is " + c.area(), 25, 130 );
64         g.drawString( "Volume is " + c.volume(), 25, 145 );
65     }
66 }

```

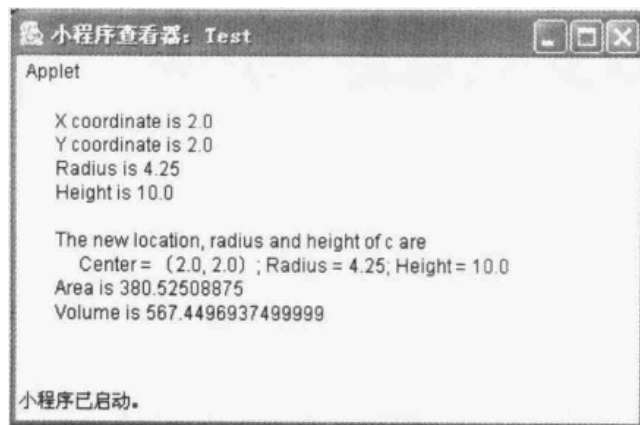


图 7.8 测试 Cylinder 类

图 7.6 显示了 Point 类。第 1 行 ~ 第 26 行是 Point 的定义。注意，Point 的实例变量为 protected 类型，因此当 Circle 类派生自 Point 类时，Circle 类的方法就能够直接引用坐标 x 和 y，而不必使用访问方法，这可以带来较好的软件性能。

图 7.6 的第 27 行 ~ 第 49 行显示了用来测试 Point 类的一个 Test applet。注意，Point 方法必须使用 getX 方法和 getY 方法来读取 protected 实例变量 x 和 y。记住，protected 实例变量只对其类的方法、其子类的方法以及处于同一个软件包中其他类的方法是可见的。

图 7.7 的第二个例子重用了图 7.6 的 Point 类定义和方法定义，因此没有再显示类定义。图 7.7 的第 1 行 ~ 第 38 行显示了 Circle 的类定义以及方法定义，第 39 行 ~ 第 65 行显示了一个 Test applet。注意，Circle 类继承了 Point 类，这表示 Circle 的 public 接口包括了 Point 的方法以及 Circle 的方法 setRadius、getRadius、area、toString 和两个 Circle 构造函数。Test applet 实例化了 Circle 类的一个对象（第 49 行），接着使用 get 方法获得关于 Circle 对象的信息，然后 Test applet 的 paint 方法再一次通过方法调用，从而间接引用了 Circle 类的 protected 数据。Test applet 接着利用 set 方法 setRadius 和 setPoint 重新设置半径和圆心的坐标。最后，Point 显示新的 Circle 对象 c，并计算和打印出它的面积。

图 7.8 的第三个例子重用了图 7.6 和图 7.7 中的 Point 类定义和 Circle 类定义。第 1 行 ~ 第 35 行显示了 Cylinder 类定义以及 Cylinder 方法定义。第 36 行 ~ 第 66 行是一个 Test applet，用来测试 Cylinder 类。注意，Cylinder 类继承了 Circle 类，这说明 Cylinder 的 public 接口包括了 Circle 方法和 Cylinder 构造函数，以及 Cylinder 方法 setHeight、getHeight、area（重写了 Circle 的 area 方法）、volume 和 toString。Test applet 实例化了 Cylinder 类的对象（第 46 行），然后利用 get 方法（第 51 行 ~ 第 54 行）获得了关于 Cylinder 对象的信息。Test applet 的 paint 方法不能再直接引用 Cylinder 类的 protected 数据。Test applet 接着利用 set 方法 setHeight、setRadius 和 setPoint 来重新设置 height、radius 以及圆柱体的坐标，然后使用 toString、area 和 volume 方法来打印 Cylinder 的属性。这个例子很好地显示了继承、定义和

引用protected实例变量。在下面的几节中,我们将指出如何以通用方式——使用多态在继承的层次中编程。数据抽象、继承以及多态是面向对象编程的关键。

## 7.10 多态简介

通过使用多态,就可能设计和实现更易于扩充的系统。如同超类对象一样,可以编写程序来通用化地处理一个层次中所有类的对象。对于在程序开发阶段不存在的类,不需要修改或只需稍加修改就能将其加入到程序的通用部分中,只要这些对象是正进行通用处理的层次的一部分即可。程序中需要修改的部分只是那些需要加入层次中的特定类的信息。我们将研究两个基本的类层次,并指出这些层次中的对象是如何以多态方式操作的。

## 7.11 类型域和 switch 语句

操作许多不同类型对象的一个办法是,使用switch语句对该对象类型的每个对象分别采取合适的动作。例如,在形状层次中,每个形状都有一个 shapeType 实例变量,switch 结构可以根据对象的 shapeType 来判断调用哪个 print 方法。

使用switch逻辑有许多问题。程序员可能在确定某一个类型时忘记了进行一个类型测试,也可能忘记了测试一个 switch 中所有可能的情况。如果一个基于 switch 的系统通过增加新类型来修改系统,那么程序员可能会忘记将新的情况插入到现有的 switch 语句中。一个类的每一次增加或删除,都需要修改系统中的每个 switch 语句;跟踪这些情况可能是费时的,并且易于出错。

正如我们将要看到的,多态编程可以免除switch逻辑。程序员可以使用Java的多态机制来自动实现等价的逻辑,因此避免了由 switch 逻辑带来的常见错误。

### 测试与调试提示 7.2

使用多态的一个有趣结果是程序的外观简洁,它们包含较少的分支逻辑,倾向于更简化的顺序代码,这种简化改善了测试、调试和程序维护。

## 7.12 动态方法绑定

设想有以下一组形状类,例如 Circle、Triangle、Rectangle 和 Square 等,这些类都派生自超类 Shape。在面向对象的编程中,这些类中的每一个都包含了自我绘图的能力。尽管每个类有它自己的 draw 方法,但每个形状的 draw 方法却大不一样。当绘制形状时,无论形状如何,最好能将这些形状都视为超类 Shape 的对象。然后在绘制形状时,我们只需简单地调用超类 Shape 的方法 draw,从而让程序动态地(即在执行时)判断使用哪个子类的 draw 方法。

### 软件工程视点 7.9

如果子类不重定义一个方法,则将简单地从它的当前超类中继承方法的定义。

如果我们使用超类引用来引用子类对象并激活 draw 方法,那么程序将会正确地动态(即在执行时)选择子类的 draw 方法,这种方式称为动态方法绑定(dynamic method binding),我们将在本章后面的实例中详细讨论。

## 7.13 final 方法和类

我们在第4章介绍过,可以将变量声明为final,表明它们在声明之后不能再修改,它们必须在声明时初始化。也可以使用final修饰符来定义方法和类。

不能在子类中重写声明为final的方法。声明为static和private的方法都隐含了final限定。由于一个final方法的定义不能再更改,因此编译器可以删除对final方法的调用,并在每个方法调用的位置使用其定义的扩展代码来替换它们,从而优化了程序。这是一种称为内联代码(inlining the code)的技术。

声明为final的类不能成为超类,即不能继承final类,final类中的所有方法隐含地都为final类型。

### 性能提示 7.2

编译器能够决定内联一个final方法调用,并且对简单的final方法也可以。内联并不破坏封装和信息隐藏(但改进了性能,因为内联去除了方法调用的负担)。

### 性能提示 7.3

流水线式处理器能够利用同时执行后续几条指令的办法来改进性能,但在这些指令后进行方法调用则是不行的。因此,内联技术(编译器可以在一个final方法上执行)能够在流水线式处理器上提高性能,因为它取消了与一个方法调用有关的越行控制(out-of-line)转移。

### 软件工程视点 7.10

不能继承声明为final的类,每个final类的方法隐含地都为final类型。

## 7.14 抽象超类和具体类

当我们认为一个类是一种类型的时候,就假定这个类型的对象将被实例化。但是,在某些情况下,对于不想实例化任何对象的程序员而言,也可以定义具有这种特点的类。这种类称为抽象类(abstract class)。因为这些类在继承的情况下是作为超类的,我们正式称其为抽象超类(abstract superclass),不能实例化抽象超类的对象。

抽象超类的惟一目的是提供合适的超类,其他类可以继承其接口或实现这些类(我们将看到每种情况的例子),可以从其中实例化对象的类称为具体类(concrete class)。

我们可以定义一个抽象超类TwoDimensionalObject和派生具体类Square、Circle和Triangle等。我们也可以定义一个抽象超类ThreeDimensionObject和派生具体类Cube、Sphere和Cylinder等。抽象超类太过于一般化,因此无法定义真正的对象;在实例化对象之前需要更加具体的类。这就是具体类要完成的工作,具体类提供了具体的内容,以便能胜任实例化对象的任务。

可以使用关键字abstract将类声明为抽象类。

### 软件工程视点 7.11

如果子类派生自一个带有abstract方法的超类,而且在子类中没有为这个abstract方法提供定义(即没有在于子类中重写这个方法),那么该方法在子类中仍为abstract类型。最后,子类也成为abstract类,而且必须显式地声明为abstract。

### 软件工程视点 7.12

将方法声明为abstract,可以为类的设计人员在类的层次中实现子类提供很大的方便。任何想从类中继承

的新类都要强制重写 abstract 方法(直接地或者从一个已重写此方法的类继承),否则这个新类将包含 abstract 方法,从而成为 abstract 类,因此不能实例化对象。

#### 软件工程视点 7.13

abstract 类也可以有实例数据,以及子类可以继承的非 abstract 方法。

#### 常见编程错误 7.10

试图实例化 abstract 类(即包含有一个或多个 abstract 方法的类)的对象是语法错误。

#### 常见编程错误 7.11

没有显式地将带有一个或多个 abstract 方法的类声明为 abstract 是语法错误。

一个层次不需要包含任何 abstract 类,但正如我们将要看到的,许多良好的面向对象的系统都有以 abstract 超类开头的层次。在某些实例中,abstract 类还构成了层次的前几层,例如形状层次。层次可以从 abstract 超类开始,在第二层我们又有两个 abstract 超类,称为 TwoDimensionalShape 和 ThreeDimensionalShape,再下一层则开始为二维形状(如 Circle 和 Square)定义具体类,以及为三维形状(如 Sphere 和 Cube)定义具体类。

## 7.15 多态的例子

我们给出了一个多态的例子。如果 Rectangle 类派生自 Quadrilateral 类,那么 Rectangle 对象就是 Quadrilateral 对象的更精确的版本。一个能在 Quadrilateral 类的对象上进行的操作(如计算周长或面积),也可以应用在 Rectangle 类的一个对象上,这类操作也能在“其他种类”的 Quadrilateral(如 Square、Parallelograms 和 Trapezoids)上进行。当从超类引用发出使用某个方法的请求时,Java 将在与对象有关的合适子类中多态地选择正确的重写方法。

假定我们有一个视频游戏,该游戏操纵许多不同的对象,包括 Martian、Venutian、Plutonian、SpaceShip、LaserBeam 等类的对象。Java 屏幕管理程序将简单地维护含有这些不同类对象的某种容器(如数组)。为了周期性地刷新屏幕,屏幕管理器将只向每个对象发出同样的消息,称为 drawYourself。每个对象将用其自身特殊的方式响应,Martian 对象将画出自己适当数目的触角,SpaceShip 对象将画出自己的亮银白色外观,LaserBeam 对象把自己画成划过屏幕的一束红光。因此,发送给不同对象的相同消息采用的形式各异,从而形成了多态。

通过这样的一个多态屏幕管理器,可以很容易地向系统添加新类型的对象。假设我们要在视频游戏中加上 Mercurian,则必须建立一个新类 Mercurian,并且保证有一个 drawYourself 方法。于是当 Mercurian 类的对象出现在容器中时,屏幕管理器不必修改屏幕。程序只是向容器中的每个对象发送消息 drawYourself,而不管对象的类型如何,因此新的 Mercurian 对象恰好“适合”。使用多态性可以在系统创建时自动添加不可预见的新对象类型,而系统则不需做任何修改(当然不包括新类型自身)。

通过使用多态性,一个方法调用可以根据接收调用的对象类型的不同而产生不同的操作,这样就赋予了程序员更好的表达能力,我们将在后而几节中看到多态的优势。

#### 软件工程视点 7.14

使用多态性,程序员可以只使用通用方法来处理对象,并让执行时的环境对于具体问题进行具体分析。程序员可以使大量的对象使用适合于自身的方式来执行任务,而不必知道这些对象的具体类型。

#### 软件工程视点 7.15

多态性推进了可扩展性：用来激活多态动作的软件和消息（即方法调用）与要送达的对象的类型无关，因此，能够响应现有消息的新类型的对象，可以不用修改基本系统就将其加入到系统中。

#### 软件工程视点 7.16

如果一个方法声明为 `final`，那么就不能在子类中重写该方法，于是方法调用不应多态地传递到这些子类的对象上。虽然方法调用可以传递到子类，但它们将以相同的方式响应而不是多态地响应。

#### 性能提示 7.4

声明为 `final` 的方法使用静态绑定而非动态绑定的方式，静态绑定比动态绑定的处理时间要少。

#### 软件工程视点 7.17

`abstract` 类为一个类层次的各个成员定义了 `public` 接口，并且包括了将在子类中定义的方法，层次中的所有类可以通过多态来使用这个相同的接口。

尽管我们不能实例化 `abstract` 超类的对象，但是能够声明对 `abstract` 超类的引用。在具体类中实例化这些对象时，这类引用可以使子类的多态操作成为可能。

让我们考虑更多的多态应用。一个屏幕管理器需要显示各种对象，甚至包括在这个屏幕管理器完成之后还将加入到系统中的新类型对象。系统可能需要显示各种形状（即超类为 `Shape`），如 `Square`、`Circle`、`Triangle`、`Rectangle` 等（这些形状类都派生自超类 `Shape`）。屏幕管理器使用超类引用（对 `Shape`）来管理要显示的所有对象。为了绘制任意对象（不考虑该对象在继承层次中出现的层次），屏幕管理器使用对象的超类引用，并简单地向该对象发出一个 `draw` 消息。`draw` 方法已经在超类 `Shape` 中声明为 `abstract`，并且已经在每个子类中重写了该方法。每个 `Shape` 对象知道如何绘制自己。屏幕管理器不必担心对象是哪一种类型或者以前是否曾经遇到过这种类型的对象，程序只需简单地告诉每个对象绘制（`draw` 方法）自身即可。

多态对于实现分层的软件系统尤其有效。例如在操作系统中，每种类型的物理设备彼此间的操作大不相同。即使这样，从设备读（`read`）或写（`write`）数据的命令仍可以有确定的一致性。发送到设备驱动程序对象的写消息，需要在设备驱动程序和该设备驱动程序操作特定类型设备的环境中分别进行解释。但是，写调用实际上同向系统中任何其他设备的写操作之间没有什么不同——只是从内存中将一些数目的字节放到一个设备上。一个面向对象的操作系统可以使用一个 `abstract` 超类来为所有的设备驱动程序提供一个接口。于是，通过继承这个 `abstract` 超类，就形成了所有操作类似的子类。由设备驱动程序提供的功能（即 `public` 接口）在 `abstract` 超类中称为 `abstract` 方法，在相应的特定类型设备驱动程序的子类中提供了这些 `abstract` 方法的实现。

在面向对象的编程中常常定义一个迭代器类（`iterator class`），以遍历一个容器（如一个数组）中的所有对象。如果用户想要打印链表中的一串对象，则可以实例化一个迭代器对象，而每次调用迭代器后都返回链表中的下一个元素。迭代器通常应用在多态编程中，用于遍历一个层次中各级对象的数组或链表，这样一个链表中的引用是超类引用（参见第 17 章以获取更多关于链表的知识）。超类对象的链表 `TwoDimensionalShape` 可以包括来自 `Square`、`Circle`、`Triangle` 等类的对象。向链表中的每个对象发送一个 `draw` 消息，并利用多态在屏幕上绘制正确的图形。

## 7.16 案例分析：一个使用多态的工资支付系统

让我们使用 `abstract` 方法和多态来完成基于雇员类型（如图 7.9 所示）的工资支付计算。我们



使用了abstract超类Employee。Employee的子类是Boss，该子类获得一份固定的周薪而不计工作的小时数；CommissionWorker获得一份基本薪水加上一份销售提成；PieceWorker的薪水根据生产件数而定；HourlyWorker的薪水根据小时计算并有加班报酬。此例中，Employee的每个子类都已声明为final，因为我们并不想再一次从它们继承。

earnings方法调用适用于所有的雇员，但是每个人收入的计算方式依赖于雇员的类，这些类皆派生自超类Employee。于是，在超类Employee中将收入声明为abstract类型，然后分别在每个子类中提供相应的earnings实现。接着，程序简单地使用超类引用来自用雇员的对象并激活earnings方法，以计算雇员的收入。在实际的支付系统中，也许会使用一个Employee引用数组中的单个元素来引用不同的Employee对象。程序简单地遍历此数组，每次处理一个元素，使用Employee引用来激活每个对象的earnings方法。

让我们考虑一个Employee类（第1行~第27行）。public方法中包括一个构造函数，它将名字和姓氏作为参数；getFirstName方法返回名字，getLastName方法返回姓氏；还有一个abstract方法——earnings。为什么这个方法是abstract类型的？因为在Employee类中提供此方法的实现没有任何意义。我们无法为一个一般化的雇员计算收入——必须首先知道雇员的类型。因此将该方法置为abstract，就是要表明我们将在每个子类中提供一个实现，但这并不在超类自身中。

Boss类（第28行~第54行）派生自Employee类。public方法包括一个构造函数，它使用名字、姓氏和周薪作为参数，并向Employee构造函数传递名字和姓氏，用来初始化子类对象中超类部分的firstName和lastName成员。setWeeklySalary方法向private实例变量weeklySalary赋一个新值，earnings方法定义了如何计算Boss的收入，toString方法形成一个包含雇员类型（即“Boss:”）、后面跟着老板名字的String。

```
1 // Fig. 7.9: Employee.java
2 // Abstract base class Employee
3
4 public abstract class Employee {
5     private String firstName;
6     private String lastName;
7
8     // Constructor
9     public Employee( String first, String last )
10    {
11        firstName = new String ( first );
12        lastName = new String( last );
13    }
14
15    // Return a copy of the first name
16    public String getFirstName()
17    { return new String( firstName ); }
18
19    // Return a copy of the last name
20    public String getLastName()
21    { return new String( lastName ); }
22
23    // Abstract method that must be implemented for each
24    // derived class of Employee from which objects
25    // are instantiated.
26    abstract double earnings();
27 }
```

```
28 // Fig. 7.9: Boss.java
29 // Boss class derived from Employee
30
31 public final class Boss extends Employee {
32     private double weeklySalary;
33
34     // Constructor for class Boss
35     public Boss( String first, String last, double s)
36     {
37         super( first, last ); // call base-class constructor
38         setWeeklySalary( s );
39     }
40
41     // Set the Boss's salary
42     public void setWeeklySalary( double s )
43     { weeklySalary = ( s > 0 ? s : 0 ); }
44
45     // Get the Boss's pay
46     public double earnings() { return weeklySalary; }
47
48     // Print the Boss's name
49     public String toString()
50     {
51         return "Boss: " + getFirstName() + ' ' +
52             getLastName();
53     }
54 }
55 // Fig. 7.9: CommissionWorker.java
56 // CommissionWorker class derived from Employee
57
58 public final class CommissionWorker extends Employee {
59     private double salary; // base salary per week
60     private double commission; // amount per item sold
61     private int quantity; // total items sold for week
62
63     // Constructor for class CommissionWorker
64     public CommissionWorker( String first, String last,
65                             double s, double c, int q)
66     {
67         super( first, last ); // call base-class constructor
68         setSalary( s );
69         setCommission( c );
70         setQuantity( q );
71     }
72
73     // Set CommissionWorker's weekly base salary
74     public void setSalary( double s )
75     { salary = ( s > 0 ? s : 0 ); }
76
77     // Set CommissionWorker's commission
78     public void setCommission( double c )
79     { commission = ( c > 0 ? c : 0 ); }
80
81     // Set CommissionWorker's quantity sold
82     public void setQuantity( int q )
83     { quantity = ( q > 0 ? q : 0 ); }
```

```
84
85     // Determine CommissionWorker's earnings
86     public double earnings()
87     { return salary + commission * quantity; }
88
89     // Print the CommissionWorker's name
90     public String toString()
91     {
92         return "Commission worker: " +
93             getFirstName() + ' ' + getLastName();
94     }
95 }
96 // Fig. 7.9: PieceWorker.java
97 // PieceWorker class derived from Employee
98 public final class PieceWorker extends Employee {
99     private double wagePerPiece;    // wage per piece output
100    private int quantity;           // output for week
101
102    // Constructor for class PieceWorker
103    public PieceWorker( String first, String last,
104                      double w, int q )
105    {
106        super( first, last ); // call base-class constructor
107        setWage( w );
108        setQuantity( q );
109    }
110
111    // Set the wage
112    public void setWage( double w )
113    { wagePerPiece = ( w > 0 ? w : 0 ); }
114
115    // Set the number of items output
116    public void setQuantity( int q )
117    { quantity = ( q > 0 ? q : 0 ); }
118
119    // Determine the PieceWorker's earnings
120    public double earnings()
121    { return quantity * wagePerPiece; }
122
123    public String toString()
124    {
125        return "Piece worker: " +
126            getFirstName() + ' ' + getLastName();
127    }
128 }
129 // Fig. 7.9: HourlyWorker.java
130 // Definition of class HourlyWorker
131
132 public final class HourlyWorker extends Employee {
133     private double wage;    // wage per hour
134     private double hours;   // hours worked for week
135
136     // Constructor for class HourlyWorker
137     public HourlyWorker( String first, String last,
138                       double w, double h )
139     {
```

```

140         super( first, last );    // call base-class constructor
141         setWage( w );
142         setHours( h );
143     }
144
145     // Set the wage
146     public void setWage( double w )
147     { wage = ( w > 0 ? w : 0 ); }
148
149     // Set the hours worked
150     public void setHours( double h )
151     { hours = ( h >= 0 && h < 168 ? h : 0 ); }
152
153     // Get the HourlyWorker's pay
154     public double earnings() { return wage * hours; }
155
156     public String toString()
157     {
158         return "Hourly worker: " +
159             getFirstName() + ' ' + getLastName();
160     }
161
162 }
163 // Fig. 7.9: Test.java
164 // Driver for Employee hierarchy
165 import java.awt.Graphics;
166 import java.applet.Applet;
167
168 public class Test extends Applet {
169     private Employee ref; // base-class reference
170     private Boss b;
171     private CommissionWorker c;
172     private PieceWorker p;
173     private HourlyWorker h;
174
175     public void init()
176     {
177         b = new Boss( "John", "Smith", 800.00 );
178         c = new CommissionWorker( "Sue", "Jones",
179                                 400.0, 3.0, 150 );
180         p = new PieceWorker( "Bob", "Lewis", 2.5, 200 );
181         h = new HourlyWorker( "Karen", "Price", 13.75, 40 );
182     }
183
184     public void paint( Graphics g )
185     {
186         ref = b; // superclass reference to subclass object
187         g.drawString( ref.toString() + " earned $" +
188                     ref.earnings(), 25, 25 );
189         g.drawString( b.toString() + " earned $" +
190                     b.earnings(), 25, 40 );
191
192         ref = c; // superclass reference to subclass object
193         g.drawString( ref.toString() + " earned $" +
194                     ref.earnings(), 25, 55 );
195         g.drawString( c.toString() + " earned $" +

```

```

196             c.earnings(), 25, 70 );
197
198     ref = p; // superclass reference to subclass object
199     g.drawString( ref.toString() + " earned $" +
200                 ref.earnings(), 25, 85 );
201     g.drawString( p.toString() + " earned $" +
202                 p.earnings(), 25, 100 );
203
204     ref = h; // superclass reference to subclass object
205     g.drawString( ref.toString() + " earned $" +
206                 ref.earnings(), 25, 115 );
207     g.drawString( h.toString() + " earned $" +
208                 h.earnings(), 25, 130 );
209 }
210 }

```

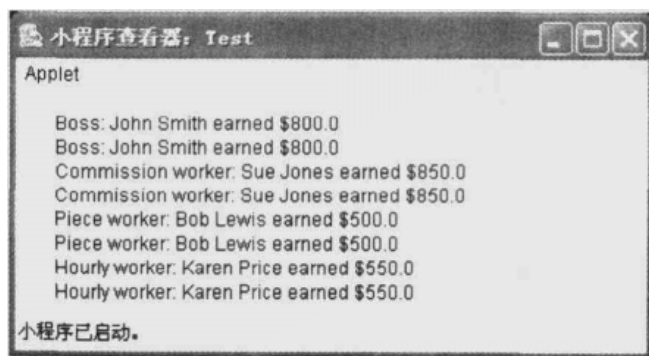


图 7.9 使用一个 abstract 超类的 Employee 类层次

CommissionWorker 类（第 55 行 ~ 第 95 行）派生自 Employee。public 方法中包含了构造函数，以名字、姓名、薪水、佣金和销售件数作为参数，并向 Employee 构造函数传递名字和姓氏。set 方法向 private 实例变量 salary、commission 和 quantity 赋值，earnings 方法定义了如何计算 CommissionWorker 的收入，toString 方法形成包括雇员类型（即 "Commission Worker:"）后跟该佣金工人姓名的一个 String。

PieceWorker 类（第 96 行 ~ 第 128 行）派生自 Employee。public 方法中包括一个构造函数，以名字、姓氏、每件付薪数以及生产件数作为参数，并向 Employee 构造函数传递名字和姓氏。set 方法向 private 实例变量 wagePerPiece 和 quantity 赋值，earnings 方法定义了如何计算 PieceWorker 的收入，toString 方法形成包括雇员类型（即 "Piece Worker:"）后跟计件工人姓名的一个 String。

HourlyWorker 类（第 129 行 ~ 第 162 行）派生自 Employee。public 方法中包括了一个构造函数，它以名字、姓氏、工资以及工作小时数作为参数，并向 Employee 构造函数传递名字和姓氏，用来初始化子类对象中超类部分的 firstName 和 lastName 成员。set 方法向 private 实例变量 wage 和 hours 赋值，earnings 方法定义了如何计算 HourlyWorker 的收入，toString 方法形成一个包含雇员类型（即 "Hourly Worker:"）后跟小时工人姓名的 String。

Test applet（第 163 行 ~ 第 210 行）开始声明了 Employee 引用 ref。每个类型的 Employee 在 Test 中的处理都类似，因此这里我们只讨论 Test 处理一个 Boss 对象的实例。

下列语句：

```
Boss b = new Boss ("John", "Smith", 800.00);
```

实例化了 Boss 类的对象 b 并提供了构造函数参数, 包括名字、姓氏及固定薪水。

在第 186 行中:

```
ref = b, //superclass reference to subclass object
```

使超类引用 ref 指向了子类对象 b。这正是我们要做的影响多态行为的工作。

第 187 行的表达式中:

```
ref.toString()
```

激活了 ref 引用的对象的 toString 方法。系统激活了该子类对象的 toString 方法——即我们称为多态的行为。这个方法的调用是一个动态方法绑定的例子——对于激活哪个方法的决定将推迟到执行时。

在第 188 行的方法调用中:

```
ref.earnings();
```

激活了由 ref 引用的对象的 earnings 方法。系统激活了该子类对象的 earnings 方法, 而不是超类的 earnings 方法, 这也是动态方法绑定的一个例子。

下面的调用:

```
b.toString();
```

利用句点成员选择运算符和特定的 Boss 对象 b 一起显式激活了 toString 方法的 Boss 版本。这是一个静态绑定的例子。因为对象 b 的类型在编译时已知, 又因为 Boss 类是一个 final 类, 所以编译器能够确定在这种情况下调用的 toString 方法只可能是 Boss 类的 toString 方法。加入该调用的目的是为了比较, 以确保使用 ref.toString() 激活的动态绑定方法的确是合适的方法。

下列语句:

```
b.earnings();
```

使用句点成员选择运算符加上特定 Boss 对象 b, 显式地激活了 earnings 方法的 Boss 版本, 这也是一个静态的例子。加入该调用的目的是为了比较, 以确保用 ref.earnings() 激活的动态绑定方法的确是合适的方法。

## 7.17 新类和动态绑定

在所有可能的类事先已知时, 多态性肯定可以工作得很好。但在向系统添加新类时, 多态性也能发挥作用。

可以使用动态方法绑定 (也叫延迟绑定) 的方法来添加新类。对于要编译的多态调用而言, 在编译时不必知道对象的类型。在执行时, 该调用利用被调用对象的方法进行匹配。

在将新对象加入系统中时, 屏幕管理程序在执行时就能处理 (不用重新编译) 新的显示对象, draw 方法调用也是一样。每个新对象自身都包含一个 draw 方法, 用来实现实际的绘图功能, 这样可以很容易地向系统中添加新的功能而影响最小, 同时也推进了软件重用。

### 性能提示 7.5

利用动态方法绑定实现的多态性可以提高系统的运行效率。

**性能提示 7.6**

利用动态绑定实现的各种多态操作也可以通过基于对象类型域的switch逻辑(手工编码)来完成。由Java编译器生成的多态代码同高效的switch逻辑编码相比,其运行效率不相上下。

## 7.18 案例分析:继承接口与实现

下一个例子(如图7.10所示)再一次考察了Point、Circle和Cylinder层次,不过现在是利用abstract超类Shape作为这个层次的顶层。Shape类有一个abstract方法getName,因此Shape是一个abstract超类。Shape包含两个其他的方法——area和volume,它们的实现都返回0值。Point从Shape继承了这些实现。因为点的面积和体积都为0,因此这是有意义的。Circle从Point继承了volume方法,但Circle提供了它自身的area方法的实现。Cylinder提供了对area(解释为圆柱体的表面积)和volume方法的自身实现。

注意,尽管Shape是一个abstract超类,但却包含了area和volume方法的实现,并且这些实现是可继承的。Shape类通过三个方法提供了一个可继承的接口,层次中的所有类都将包含这些方法。Shape类也提供了一些在层次的前几层子类中用到的一些实现。本实例强调一个子类能从一个超类中继承接口或实现。

**软件工程视点 7.18**

面向实现继承的层次趋向于在层次中提高它们的功能性——每个新子类继承一个或多个在超类中定义的方法,并且使用超类的定义。

**软件工程视点 7.19**

面向接口继承的层次趋向于在层次中降低它们的功能性——超类定义了一个或几个方法,它们应由层次中的每个对象以同样的方式调用(即它们有相同的特征),但是单个子类提供了自己的方法实现。

---

```
1      // Fig. 7.10: Shape.java
2      // Definition of abstract base class Shape
3
4      public abstract class Shape {
5          public double area() { return 0.0; }
6          public double volume() { return 0.0; }
7          public abstract String getName();
8      }
9      // Fig. 7.10: Point.java
10     // Definition of class Point
11
12     public class Point extends Shape {
13         protected double x, y; // coordinates of the Point
14
15         // constructor
16         public Point( double a, double b ) { setPoint( a, b ); }
17
18         // Set x and y coordinates of Point
19         public void setPoint( double a, double b )
20         {
21             x = a;
22             y = b;
23         }
24     }
```

```

25     // get x coordinate
26     public double getX() { return x; }
27
28     // get y coordinate
29     public double getY() { return y; }
30
31     // convert the point into a String representation
32     public String toString()
33     { return "(" + x + ", " + y + ")"; }
34
35     // return the class name
36     public String getName() { return "Point"; }
37 }
38 // Fig. 7.10: Circle.java
39 // Definition of class Circle
40
41 public class Circle extends Point { // inherits from Point
42     protected double radius;
43
44     // no-argument constructor
45     public Circle()
46     {
47         super( 0, 0 ); // call the base class constructor
48         setRadius( 0 );
49     }
50
51     // Constructor
52     public Circle( double r, double a, double b )
53     {
54         super( a, b ); // call the base class constructor
55         setRadius( r );
56     }
57
58     // Set radius of Circle
59     public void setRadius( double r )
60     { radius = ( r >= 0 ? r : 0 ); }
61
62     // Get radius of Circle
63     public double getRadius() { return radius; }
64
65     // Calculate area of Circle
66     public double area() { return 3.14159 * radius * radius; }
67
68     // convert the Circle to a String
69     public String toString()
70     { return "Center = " + super.toString() +
71         " ; Radius = " + radius; }
72
73     // return the class name
74     public String getName() { return "Circle"; }
75 }
76 // Fig. 7.10: Cylinder.java
77 // Definition of class Cylinder
78
79 public class Cylinder extends Circle {
80     protected double height; // height of Cylinder

```



```
81
82 // Cylinder constructor calls Circle constructor
83 public Cylinder( double h, double r, double a, double b )
84 {
85     super( r, a, b ); // call base-class constructor
86     setHeight( h );
87 }
88
89 // Set height of Cylinder
90 public void setHeight( double h )
91 { height = ( h >= 0 ? h : 0 ); }
92
93 // Get height of Cylinder
94 public double getHeight() { return height; }
95
96 // Calculate area of Cylinder (i.e., surface area)
97 public double area()
98 {
99     return 2 * super.area() +
100         2 * 3.14159 * radius * height;
101 }
102
103 // Calculate volume of Cylinder
104 public double volume() { return super.area() * height; }
105
106 // Convert a Cylinder to a String
107 public String toString()
108 { return super.toString() + "; Height = " + height; }
109
110 // Return the class name
111 public String getName() { return "Cylinder"; }
112 }
113 // Fig. 7.10: Test.java
114 // Driver for point, circle, cylinder hierarchy
115 import java.awt.Graphics;
116 import java.applet.Applet;
117
118 public class Test extends Applet {
119     private Point point;
120     private Circle circle;
121     private Cylinder cylinder;
122     private Shape arrayOfShapes[];
123
124     public void init()
125     {
126         point = new Point( 7, 11 );
127         circle = new Circle( 3.5, 22, 8 );
128         cylinder = new Cylinder( 10, 3.3, 10, 10 );
129
130         arrayOfShapes = new Shape[ 3 ];
131
132         // aim arrayOfShapes[ 0 ] at subclass Point object
133         // aim arrayOfShapes[ 1 ] at subclass Circle object
134         // aim arrayOfShapes[ 2 ] at subclass Cylinder object
135         arrayOfShapes[ 0 ] = point;
136         arrayOfShapes[ 1 ] = circle;
```

```

137     arrayOfShapes[ 2] = cylinder;
138 }
139
140 public void paint( Graphics g )
141 {
142     g.drawString( point.getName() + ": " +
143                 point.toString(), 25, 25 );
144
145     g.drawString( circle.getName() + ": " +
146                 circle.toString(), 25, 40 );
147
148     g.drawString( cylinder.getName() + ": " +
149                 cylinder.toString(), 25, 55 );
150
151     int yPos = 85;
152
153     // Loop through arrayOfShapes and print the name,
154     // area, and volume of each object.
155     for ( int i = 0; i < 3; i++ ) {
156         g.drawString( arrayOfShapes[ i].getName() +
157                     ": " + arrayOfShapes[ i].toString(),
158                     25, yPos );
159         yPos += 15;
160         g.drawString( "Area = " +
161                     arrayOfShapes[ i].area(), 25, yPos );
162         yPos += 15;
163         g.drawString( "Volume = " +
164                     arrayOfShapes[ i].volume(), 25, yPos );
165         yPos += 30;
166     }
167 }
168 }

```



图 7.10 Shape、Point、Circle 和 Cylinder 的层次

超类 Shape (第 1 行 ~ 第 8 行) 包括 3 个 public 方法, 没有包含任何数据。getName 方法为 abstract 类型, 所以在每个子类中都重写该方法。area 和 volume 方法则定义为返回 0.0。当这些方法适合子

那些有不同的 area 计算或不同的 volume 计算的类时,则将在子类中重写这些方法。

Point 类(第9行~第37行)派生自 Shape 类。一个 Point 对象的面积为 0.0, 体积为 0.0, 所以在此处未重写超类方法 area 和 volume, Point 类将继承 Shape 类的数据。getName 方法是超类中 abstract 方法的一个实现。其他方法包括 setPoint, 用以给 x 和 y 坐标赋新值, 而 getX 和 getY 返回 Point 的 x 和 y 坐标。

Circle 类(第38行~第75行)派生自 Point 类。Circle 的体积为 0.0, 所以在此处未重写超类方法 volume——Circle 类继承从 Shape 类继承而来的 Point 类。Circle 的面积不同于一个点, 所以在这个类中重写 area 方法。getName 方法是超类中 abstract 方法的一个实现, 如果在此处未重写该方法, 则将继承 Point 版本的 getName。其他方法包括给一个 Circle 的 radius (半径) 赋值的 setRadius 方法, 以及返回一个 Circle 的 radius 的 getRadius 方法。

Cylinder 类(第76行~第112行)派生自 Circle, Cylinder 的面积和体积不同于 Circle 类, 于是在此类中重写 area 和 volume 方法。getName 方法是超类中 abstract 方法的一个实现, 如果在此处未重写该方法, 则将继承 Circle 版本的 getName 方法。其他方法包括给一个 Cylinder 的 height (高度) 赋值的 setHeight 方法, 还有返回一个 Cylinder 的 height 的 getHeight 方法。

Test applet (第113行~第168行)一开始就初始化 Point 对象 point、Circle 对象 circle 和 Cylinder 对象 cylinder (第126行~128行)。接着, 实例化数组 arrayOfShapes (第130行), 这个超类使用数组来引用已实例化的子类对象。然后, 将一个 point 对象的引用赋给数组元素 arrayOfShapes[0], 将一个 circle 对象的引用赋给数组元素 arrayOfShapes[1], 将一个 cylinder 对象的引用赋给数组元素 arrayOfShapes[2] (第135行~第137行)。输出各个对象, 以说明都已正确地初始化了对象 (第141行~第149行), 此时分别激活 getName 方法和 toString 方法。然后, 使用 for 结构遍历 arrayOfShapes, 并且在每一次循环中执行下列调用:

```
arrayOfShapes[ i ].getName ( )  
arrayOfShapes[ i ].toString ( )  
arrayOfShapes[ i ].area ( )  
arrayOfShapes[ i ].volume ( )
```

上面的每一个方法调用都将根据 arrayOfShapes[i] 当前所指的對象而分別激活。在正确地激活輸出顯示方法后, 首先輸出字符串 "Point:" 以及 point 對象的坐標, 其面積和體積都為 0。接着, 輸出字符串 "Circle:" 以及 circle 對象的坐標和半徑; 計算對象 circle 的面積, 其體積為 0。最后, 輸出字符串 "Cylinder:"、cylinder 對象的坐標、cylinder 對象的半徑以及 cylinder 的體積。所有的 getName、toString、area 和 volume 方法調用都是在運行時利用動態綁定的方式解決的。

## 7.19 基本类型的类型包装类

每一个基本类型都有一个类型包装类 (type wrapper class), 这些类称为 Integer、Float、Char、Long、Double 和 Boolean。使用类型包装类时, 能够像 Object 类的对象一样操作基本类型, 我们将要开发或重用的许多类都会操作并共享 Object。这些类不能操作基本类型的变量, 但是它们能操作类型包装的对象, 因为每个类最终都派生自 Object 类。每一个数字类都继承自 Number 类, 包括 Integer、Float、Long 和 Double。每个类型包装都声明为 final, 因此它们的方法隐含地都为 final, 并且不能重写。我们将在第 17 章和第 18 章介绍数据结构时使用类型包装类。

## 小结

- 面向对象编程的关键功能之一就是**通过继承来达到软件重用的目的**。
- 通过继承,一个新类继承了先前定义在超类中的实例变量和方法。在这种情况下,称新类为一个子类。
- 如果使用单继承,则类派生自一个超类;如果使用多继承,则子类可以继承多个(可能不相关的)超类。Java并不支持多继承,但是提供了接口的概念(将在第13章讨论),接口提供了继承的优点且不会产生相关的问题。
- 子类一般有自己的实例变量和方法,所以子类一般要大于它的超类,子类比其超类更加细化,一般代表较少的对象。
- 子类不能访问其超类的 `private` 成员,但可以访问 `public`、`protected` 和软件包访问成员;子类必须在超类的软件包中才能访问软件包访问的超类成员。
- 子类的构造函数通常先调用其超类的构造函数(显式或隐式)来创建和初始化子类中的超类成员。
- 继承使软件的可重用性成为可能,重用可以节约开发时间并鼓励使用预先证明过和调试过的高质量软件。
- 将来,大多数的软件将由标准化的可重用构件创建,就像如今的硬件一样。
- 子类的对象可以看成是其对应超类的对象,但反过来却不正确。
- 超类同它的子类存在于同一个层次关系中。
- 当在继承机制中使用类时,该类要么作为超类,向其他类提供属性和行为;要么作为子类,继承一些属性和行为。
- 一个继承层次在一个特定系统的物理限制内可以任意深,但大部分继承层次只有几层。
- 继承对于理解和管理系统的复杂性是有用的。随着软件变得日益复杂,Java通过继承和多态提供了支持层次结构的机制。
- `protected` 访问是介于 `public` 访问和 `private` 访问之间的中间保护层次。一个超类的 `protected` 成员可由该超类的方法访问,也可以由子类的方法和在同一个软件包内的类方法访问;此外,再无其他的方法能够访问超类的 `protected` 成员。
- 超类可以是子类的直接超类,也可以是子类的间接超类。直接超类是子类显式继承的类,间接超类是子类继承自类层次树几层之外的类。
- 当超类成员对子类不合适时,程序员可以在子类中重写这个成员。
- 区分“is a”(是)关系和“has a”(有)关系是很重要的。在“has a”关系中,一个类对象将另一个类对象的引用作为其成员。而在“is a”关系中,也可以将子类类型的对象作为超类类型的对象。“has a”是继承,“has a”是复合。
- 可以将子类对象赋值给超类引用。这种赋值是有意义的,因为该子类有对应于每个超类成员的成员。
- 子类对象的引用可以隐式转换成对超类对象的引用。
- 使用显式类型转换将超类引用转换成子类引用是可能的。如果转换目标不是子类对象,则会抛出 `ClassCastException` 异常。
- 超类定义了共性,所有自一个超类派生的类都具有这个超类的功能。在面向对象的设计过程中,设计者寻找共性并将其提取出来以组成超类。子类可以定制比超类更多的功能。

- 阅读一组子类的声明可能会引起混淆,因为在子类声明中并未列出继承的超类成员,但在子类中的确提供了这些成员。
- 对于子类对象而言,总是先显式或隐式地调用超类构造函数,接着才调用子类构造函数。
- 使用多态性使得设计和实现更易于扩展的系统成为可能,用户可以提前写出程序,以处理在程序开发时还不存在的各类对象。
- 多态编程可以不再需要 switch 逻辑,从而避免与 switch 逻辑有关的错误。
- 在超类中声明抽象方法时,应在方法的定义前加上关键字 `abstract`。
- 如果需要,子类可以提供自己的超类方法的实现;如果没有定义,则将应用超类的实现。
- 在许多情况下,需要定义一些程序员不用来实例化任何对象的类。这些类称为抽象类。因为这些类只用做超类,所以其正式的名称为抽象超类。不可以实例化抽象类的对象。
- 可以实例化对象的类称为具体类。
- 类只需将其方法声明为 `abstract` 就可以成为抽象类,这样的类必须显式地声明为抽象的。
- 如果子类派生自一个带有 `abstract` 方法的超类,而在子类中没有提供对该 `abstract` 方法的定义,那么这个方法在子类中仍然为 `abstract` 类型。结果,子类也将成为 `abstract` 类(不能实例化任何对象)。
- 对于一个通过超类引用来使用方法的请求,Java 将在与该对象相关的类中选择正确的重写方法。
- 通过使用多态,方法调用过程可以根据调用对象的不同类型而激发不同动作。
- 尽管我们不能实例化 `abstract` 超类的对象,但是可以声明对 `abstract` 超类的引用。这类引用可用于子类对象的多态操作,前提是这类对象是从具体类实例化而来的。
- 可以将新类型的类有规则地加入系统中。通过动态方法绑定(也叫延迟绑定),可以将新类包含到系统中。对于要编译的一个方法而言,一个对象的类型在编译时是不需要知道的。在执行时,选择接收对象的相应方法。
- 通过动态方法绑定,在执行时对一个方法的调用将传递到接收调用对象的类的相应方法上。
- 如果超类提供了方法,则子类可以重写此方法,但不一定必须重写,这样子类就可以使用方法的超类版本。

## 术语

`abstract class`   `abstract` 类  
`abstract method`   `abstract` 方法  
`abstract superclass`   `abstract` 超类  
`abstraction`   抽象  
`base class`   基类  
`Boolean class`   `Boolean` 类  
`Char class`   `Char` 类  
`class hierarchy`   类层次  
`client of a class`   类的客户  
`composition`   复合  
`direct superclass`   直接超类

`Double class`   `Double` 类  
`dynamic method binding`   动态方法绑定  
`extends`   扩展(继承)  
`extensibility`   可扩展性  
`final class`   `final` 类  
`final instance variable`   `final` 实例变量  
`final method`   `final` 方法  
`"friendly" access`   “友好”的访问  
`garbage collection`   无用单元回收  
`"has a" relationship`   “has a”关系  
`hierarchical relationship`   层次关系

|                                  |              |                                  |                 |
|----------------------------------|--------------|----------------------------------|-----------------|
| implementation inheritance       | 实现继承         | override a method                | 重写 (覆盖) 方法      |
| implicit reference conversion    | 隐式引用转换       | override an abstract method      | 重写 abstract 方法  |
| indirect superclass              | 间接超类         | polymorphism                     | 多态性             |
| infinite recursion error         | 无限递归错误       | overriding vs. overloading       | 重写与重载           |
| inheritance                      | 继承           | protected member of a class      | 类的 protected 成员 |
| inheritance hierarchy            | 继承层次         | reference to an abstract class   | 对 abstract 类的引用 |
| Integer class                    | Integer 类    | single inheritance               | 单继承             |
| interface                        | 接口           | software reusability             | 软件可重用性          |
| interface inheritance            | 接口继承         | standardized software components | 标准化软件构件         |
| "is a" relationship              | "is a" 关系    | static binding                   | 静态绑定            |
| "knows a" relationship           | "knows a" 关系 | subclass                         | 子类              |
| late binding                     | 延迟绑定         | subclass constructor             | 子类构造函数          |
| Long class                       | Long 类       | subclass reference               | 子类引用            |
| member access control            | 成员访问控制       | super                            |                 |
| member object                    | 成员对象         | superclass                       | 超类              |
| method overriding                | 方法重写         | superclass constructor           | 超类构造函数          |
| method table                     | 方法表          | superclass reference             | 超类引用            |
| multiple inheritance             | 多继承          | switch logic                     | switch 逻辑       |
| Number class                     | Number 类     | this                             |                 |
| Object class                     | Object 类     | type wrapper class               | 类型包装类           |
| object-oriented programming(OOP) | 面向对象编程 (OOP) | "use a" relationship             | "use a" 关系      |

## 自测练习

### 7.1 填空:

- 如果 Alpha 类继承自 Beta 类, 则 Alpha 类称为 \_\_\_\_\_ 类, Beta 类称为 \_\_\_\_\_ 类。
- 继承使 \_\_\_\_\_ 成为可能, 它节省了开发时间, 鼓励使用预先证明过的和高质量的软件构件。
- 可以认为一个 \_\_\_\_\_ 类的对象是其相应的 \_\_\_\_\_ 类的对象。
- 4 个成员访问说明符是 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 类间的 "has a" 关系表示 \_\_\_\_\_, 类间的 "is a" 关系表示 \_\_\_\_\_。
- 使用多态有助于消除 \_\_\_\_\_ 逻辑。
- 如果一个类包含了一个或多个 abstract 方法, 则其就是一个 \_\_\_\_\_ 类。
- 在编译时解决的方法调用称为 \_\_\_\_\_ 绑定。
- 在运行时解决的方法调用称为 \_\_\_\_\_ 绑定。

### 7.2 填空:

- 一个子类可以通过在方法调用前加上 \_\_\_\_\_ 来调用其超类的任意方法。
- 一个超类一般代表的对象数量要大于其子类所代表的数量 (判断对错)。
- 一个子类一般比其超类封装的功能性要少 (判断对错)。

## 自测练习答案

- 7.1 a)子,超。b)软件可重用性。c)子,超。d)public, protected, private, 软件包访问。e)复合,继承。f)switch。g)abstract。h)静态。i)动态。
- 7.2 a)super。b)对。c)错。

## 练习

- 7.3 考虑 bicycle 类。假定读者了解自行车的一些常见部件,请给出一个类层次,其中 bicycle 是从其他类继承而来的,其他类也是从上一层继承而来的。讨论 bicycle 类中各个对象的实例化。讨论其他密切相关的子类对 bicycle 的继承。
- 7.4 解释下列术语:单继承、多继承、接口、超类和子类。
- 7.5 讨论为什么将超类引用强制转换成子类引用存在潜在的危险。
- 7.6 区分单继承和多继承。为什么 Java 不支持多继承? Java 如何实现多继承的优点?
- 7.7 判断对错:子类一般小于其超类。
- 7.8 判断对错:一个子类对象也是该子类的超类的对象。
- 7.9 一些程序员不喜欢使用 protected 访问,因为它破坏了信息隐藏。讨论在超类中使用 protected 访问和 private 访问各自的优点。
- 7.10 许多利用继承编写的程序可以使用复合来解决,反之亦然。在 Point、Circle 和 Cylinder 类层次的环境中讨论这些方法的相对优点。使用复合重写图 7.10 的程序(及支持类),完成之后,重新评价在 Point、Circle 和 Cylinder 问题中,以及在一般面向对象的程序中两者所用方式的相对优点。
- 7.11 将图 7.10 的 Point、Circle 和 Cylinder 程序改写为一个 Point、Square 和 Cube 程序。使用两种方式完成——一种利用继承,另一种利用复合。
- 7.12 在本章中我们讨论过,“当一个超类成员对一个子类不合适时,可以在子类中通过一个合适的实现来重写该成员。”如果完成了这一点,那么“子类是一个超类对象”这一关系仍存在吗?为什么?
- 7.13 研究图 7.2 的继承层次。对于每个类,指出与该层次一致的一些公共属性和行为。加入其他一些类(即 UndergraduateStudent、GraduateStudent、Freshman、Sophomore、Junior 和 Senior 等来丰富这个层次)。
- 7.14 为 Quadrilateral、Trapezoid、Parallelogram、Rectangle 和 Square 类编写一个继承层次。使用 Quadrilateral 作为层次的超类。使该层次尽可能得深(即包含有许多层)。Quadrilateral 的 private 数据应当包括对应 Quadrilateral 的 4 个顶点的一对(x,y)坐标。编写一个程序,实例化并显示每个类的对象(在第 9 章将介绍如何使用 Java 的绘图功能)。
- 7.15 编写出能够想到的形状,包括二维和三维图形,将这些形状组成一个形状层次。层次中应有超类 Shape,然后从 Shape 派生出 TwoDimensionalShape 和 ThreeDimensionalShape 类。一旦建成了该层次,就定义层次中的各个类。我们将在练习中使用这个层次来处理作为超类 Shape 对象的所有形状。
- 7.16 解释多态是如何进行“泛化”而不是“特化”编程的,讨论“泛化”编程的主要优点。
- 7.17 讨论使用 switch 逻辑编程的问题,解释为什么多态可以有效地替代 switch 逻辑。

- 7.18 区分静态绑定和动态绑定。
- 7.19 区分继承接口和继承实现。用于继承接口的层次与用于继承实现的层次有什么不同?
- 7.20 区分非 abstract 方法和 abstract 方法。
- 7.21 判断对错: 所有在 abstract 超类中的方法都必须声明为 abstract。
- 7.22 为本章开始部分讨论的 Shape 层次提出一个或多个 abstract 超类层 (第一层应包括 Shape, 第二层包括 TwoDimensionalShape 和 ThreeDimensionalShape 类)。
- 7.23 多态性是如何推进可扩展性的?
- 7.24 假设要开发一个飞行模拟器, 它将给出精细的图形输出。解释为什么多态编程特别适合这类问题。
- 7.25 使用本章的 Shape 类继承层次, 开发一个基本图形软件包。只能使用二维形状, 如正方形、矩形、三角形和圆形。这个软件包需要与用户进行交互, 让用户定义绘制每个形状的位置、大小和填充字符。用户可以定义同一个形状的许多项。创建每个形状时, 将每个新 Shape 对象的 Shape 引用放入一个数组中, 每个类都有自己的 draw 方法。编写一个多态屏幕管理程序, 遍历此数组并向数组中的每个对象发出 draw 消息, 从而形成一幅屏幕影像。每次用户定义一个附加的形状时都要重画一次屏幕。
- 7.26 修改图 7.9 中的工资支付系统, 向 Employee 类添加 private 实例变量 birthDate (Date 对象) 和 departmentCode (int 对象)。假定工资支付系统每个月处理一次。然后, 当程序为每个 Employee 计算工资时 (使用多态方式), 如果这个月有 Employee 过生日, 就向他的工资总额中加入 \$1 000。
- 7.27 在练习 7.15 中, 我们开发了一个 Shape 类层次并在层次中定义了各种类。修改此层次, 使得 Shape 类成为包含此层次接口的 abstract 超类。从 Shape 类派生 TwoDimensionalShape 和 ThreeDimensionalShape, 并且这些类应当为 abstract。使用一个 abstract print 方法来输出每个类的类型和维数。还要包括 area 和 volume 方法, 可以在层次中为每个具体类的对象完成这些计算。编写一个驱动程序, 测试该 Shape 类层次。



## 第8章 字符串和字符

### 教学目标

- 创建并操作 String 类的对象
- 创建并操作 StringBuffer 类的对象
- 创建并操作 Character 类的对象
- 利用 StringTokenizer 对象，将 String 对象分成称为标记的单个组件

### 8.1 简介

在本章中，我们引入了 Java 的字符串和字符处理功能。这里讨论的技术既适合于开发文本编辑器、字处理器、排版软件、电脑打字系统，也适合开发其他的文本处理软件。我们在本书中已经提供了几个字符串处理系统，在本章我们将详细讨论 Java.lang 软件包中的 String 类、StringBuffer 类和 Character 类，以及 Java.util 软件包中的 StringTokenizer 类。

### 8.2 字符和字符串的基础

字符是构造 Java 源程序块的基础。每个程序都由一系列字符组成，并形成有意义的组合，然后作为完成一个任务的指令序列，由计算机对其进行解释。程序中可以包含字符常量，字符常量就是表示单引号内字符的一个整数值。正如我们前面讲到的，一个字符常量的值是该字符在标准字符集（Unicode）中的整数值。例如，'z' 代表 z 的整数值，'\n' 代表换行符的整数值。

由一系列字符组成的字符串可以作为一个单位进行处理。字符串可以包含字母、数字和各种特殊字符，如 +、-、\*、/、\$ 等。Java 的字符串就是 String 类的对象，可以将 Java 中的字符串直接量或字符串常量（它们常称为匿名 String 对象）写成双引号中的字符序列，如下所示：

|                          |          |
|--------------------------|----------|
| "John Q.Doe"             | ( 姓名 )   |
| "9999Main Street"        | ( 街道地址 ) |
| "Waltham, Massachusetts" | ( 城市和州 ) |
| " ( 201 ) 555-1212 "     | ( 电话号码 ) |

Java 将有相同内容的所有匿名 String 视为一个有多个引用的匿名 String 对象。一个 String 可在声明中赋给一个 String 引用。下列声明：

```
String color = " blue ";
```

将一个 String 引用 color 初始化为对匿名 String 对象 "blue" 的引用。

### 8.3 String 构造函数

String 类为以各种方式初始化 String 对象而提供了 7 个构造函数，如图 8.1 所示。所有的构造函数都用于 StringConstructor applet 的 init 方法。下列语句：

```
s1 = new String ( );
```

初始化一个新的 String 对象，并用 String 类的默认构造函数将其赋值给引用 s1。新的 String 对象没有包含任何字符且长度为 0。

```

1 // Fig. 8.1: StringConstructors.java
2 // This program demonstrates the String class constructors.
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class StringConstructors extends Applet {
7     char charArray[] = { 'b', 'a', 'r', 't', 'h', ' ',
8                          'd', 'a', 'y' };
9     byte byteArray[] = { 'n', 'e', 'w', ' ',
10                        'y', 'e', 'a', 'r' };
11     StringBuffer buffer;
12     String s, s1, s2, s3, s4, s5, s6, s7;
13
14     public void init()
15     {
16         s = new String( "hello" );
17         buffer = new StringBuffer();
18         buffer.append( "Welcome to Java Programming!" );
19
20         // use the String constructors
21         s1 = new String();
22         s2 = new String( s );
23         s3 = new String( charArray );
24         s4 = new String( charArray, 6, 3 );
25         s5 = new String( byteArray, 0, 4, 4 );
26         s6 = new String( byteArray, 0 );
27         s7 = new String( buffer );
28     }
29
30     public void paint( Graphics g )
31     {
32         g.drawString( "s1 = " + s1, 25, 25 );
33         g.drawString( "s2 = " + s2, 25, 40 );
34         g.drawString( "s3 = " + s3, 25, 55 );
35         g.drawString( "s4 = " + s4, 25, 70 );
36         g.drawString( "s5 = " + s5, 25, 85 );
37         g.drawString( "s6 = " + s6, 25, 100 );
38         g.drawString( "s7 = " + s7, 25, 115 );
39     }
40 }

```

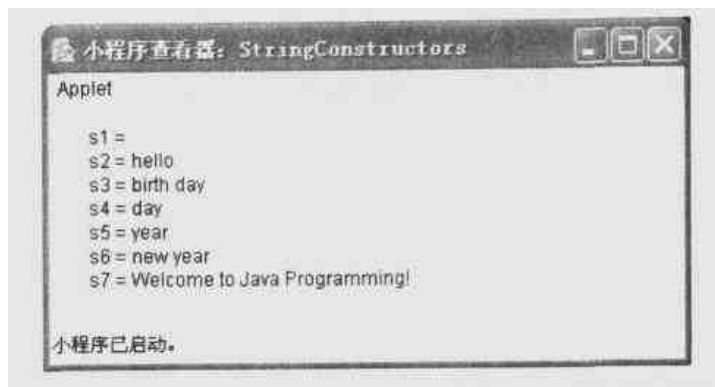


图 8.1 示例 String 类的构造函数

下列语句:

```
s2 = new String(s);
```

实例化了一个新的 String 对象, 并用 String 类的复制构造函数将其赋值给引用 s2。新的 String 对象包含了 String 对象 s 中的字符副本, 对象 s 作为参数递给构造函数。

下列语句:

```
s3 = new String ( charArray );
```

实例化了一个新的 String 对象, 并用 String 类的以一个字符数组为参数的构造函数将其赋值给引用 s3, 新的 String 对象包含了一份该数组中字符的副本。

下列语句:

```
s4 = new String ( charArray, 6, 3);
```

实例化了一个新的 String 对象, 并用 String 类的以一个字符数组和两个整数为参数的构造函数将其赋值给引用 s4。第二个参数说明了要复制的字符在数组中的起始位置 (offset)。第三个参数说明了要从数组中复制的字符个数 (count)。新的 String 对象包含了该数组中指定字符的一份副本。如果作为参数的 String 对象引用该字符数组边界之外的一个元素, 那么将抛出 StringIndexOutOfBoundsException 异常, 关于异常的详细内容请参见第 12 章。

下列语句:

```
s5 = new String ( byteArray, 0, 4, 4 );
```

实例化了一个新的 String 对象, 并用 String 类的以一个字节数组和三个整数为参数的构造函数将该对象赋给引用 s5。第三个参数和第四个参数分别定义了 offset 和 count, 第二个参数定义了 hibyte 值。Java 中的字符都是两个字节的 Unicode 字符, 但是, 一个字符数组中的每个元素都只占一个字节。hibyte 值同每个数组元素组合在一起, 创建一个双字节的字符, 它可作为 String 的一部分存储。在组成的双字节中, byte 数组中的原始 byte 值称为低位字节 (low-order byte), 而 hibyte 则称为高位字节 (high-order byte)。对于英语字符 (相当于 ASCII 字符集), hibyte 的值一般为 0。新的 String 对象包含了该数组中指定字节的一份副本。如果参数 offset 和 count 形成了对字符数组边界之外的一个元素的访问, 那么会抛出 StringIndexOutOfBoundsException 异常。

下列语句:

```
s6 = newString ( byteArray, 0 );
```

使用String类的以一个byte数组和一个整数作为参数的构造函数，实例化了一个新的String对象，并将其赋给引用s6。构造函数的第二个参数是hibyte值。新的String对象包含了该数组中字节的一份副本。

下列语句：

```
s7 = newString ( buffer );
```

使用String类的以一个StringBuffer为参数的构造函数，实例化了一个新的String对象，并将其赋值给引用s7。新的String对象包含了StringBuffer中字符的一份副本。一个StringBuffer是一个动态可变大小的字符串。下面两行：

```
buffer=new StringBuffer ( );
buffer.append( " Welcome to Java Programming " );
```

创建了StringBuffer类的一个新对象，将其赋给StringBuffer的引用buffer，并使用StringBuffer的方法append向对象buffer添加字符串“Welcome to Java Programming!”。我们将在本章后面的内容中详细讨论StringBuffer。

## 8.4 String方法：length、charAt、getChars、getBytes

本节将介绍String的方法length、charAt、getChars和getBytes，这些方法都用在applet String Misc的paint方法（如图8.2所示）中。下列语句：

```
g.drawString ( " Length of s1 :"+ s1.length ( ), 25, 40 );
```

使用String方法length来确定String s1中字符的个数。如同数组一样，String通常知道它们自己的大小。但是与数组不同的是，String没有一个length实例变量来指明一个String中的元素个数。

```

1      // Fig. 8.2: StringMisc.java
2      // This program demonstrates the length, charAt, getChars, and
3      // getBytes methods of the String class.
4      //
5      // Note: Methods getChars and getBytes require a starting point
6      // and ending point in the String. The starting point is the
7      // actual subscript from which copying starts. The ending point
8      // is one past the subscript at which the copying ends.
9      import java.awt.Graphics;
10     import java.applet.Applet;
11
12     public class StringMisc extends Applet {
13         String s1;
14         char charArray [ ];
15         byte byteArray [ ];
16
17         public void init()
18         {
19             s1 = new String( "hello there" );
20             charArray = new char [ 5 ];
21             byteArray = new byte [ 5 ];
22         }
23

```

```
24     public void paint( Graphics g )
25     {
26         // output the string
27         g.drawString( "s1: " + s1, 25, 25 );
28
29         // test the length method
30         g.drawString( "Length of s1: " + s1.length(), 25, 40 );
31
32         // loop through the characters in s1 and display reversed
33         g.drawString( "The string reversed is: ", 25, 55 );
34         int xPosition = 155;
35
36         for ( int i = s1.length() - 1; i >= 0; i-- ) {
37             g.drawString( String.valueOf( s1.charAt( i ) ),
38                         xPosition, 55 );
39             xPosition += 10;
40         }
41
42         // copy characters from string into char array
43         s1.getChars( 0, 5, charArray, 0 );
44         g.drawString( "The character array is: ", 25, 70 );
45         g.drawChars( charArray, 0, charArray.length, 158, 70 );
46
47         // copy characters from string into byte array
48         s1.getBytes( 6, 11, byteArray, 0 );
49         g.drawString( "The byte array is: ", 25, 85 );
50         g.drawBytes( byteArray, 0, byteArray.length, 130, 85);
51     }
52 }
```

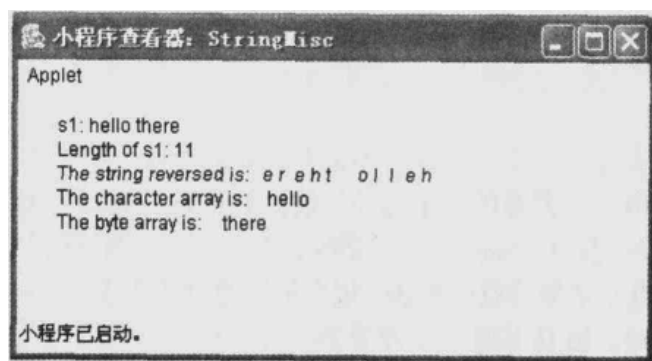


图 8.2 String 类的字符操作方法

#### 常见编程错误 8.1

试图使用一个称为 `length` 的实例变量 (即 `s1.length`) 来判断一个 `String` 类的长度是一种语法错误, 必须使用 `String` 的方法 `length` (即 `s1.length()`)。

第 36 行的 `for` 结构反向显示了 `String s1` 中的字符。`String` 方法 `charAt` 用于在 `String` 中的一个特定位置选择字符。`charAt` 方法接收一个整型参数作为位置序号 (或下标), 并返回该位置上的字符。如同数组一样, 认为 `String` 的第一个元素在位置 0 上。如果作为参数的下标超出了 `String` 的边界, 那么会抛出 `StringIndexOutOfBoundsException` 异常。

### 常见编程错误 8.2

试图访问一个在 `String` 边界之外的字符（即下标小于 0 或大于等于 `String` 的长度）会抛出 `StringIndexOutOfBoundsException` 异常。

下列语句：

```
s1.getChars (0 , 5 , charArray , 0 );
```

使用 `String` 方法 `getChars` 把 `String` 中的字符复制到字符数组。第一个参数是 `String` 中要复制的字符的开始下标。第二个参数是要从 `String` 中复制的最后字符后面的位置下标。第三个参数是放置已复制字符的字符数组。最后一个参数是已复制字符在字符数组中放置的开始下标。前面语句的操作结果可以用下列语句显示出来：

```
g.drawChars ( charArray, 0 , charArray.length , 158 , 70 );
```

上述语句使用 `Graphics` 方法 `drawChars` 来输出该字符数组。第二个参数指明了要输出的第一个字符的下标，第三个参数指明了要输出的最后一个字符后面的位置下标。

下列语句：

```
s1.getBytes( 6, 11 , byteArray , 0 );
```

使用 `String` 方法 `getBytes` 把 `String` 中的字符复制到 `byte` 数组，第一个参数是 `String` 中要复制的字符的开始下标，第二个参数是 `String` 中要复制的最后一个字符的下标，第三个参数是放置已复制字符的 `byte` 数组，最后一个参数是已复制字符在字符数组中的开始下标。只有字符的低位复制到了 `byte` 数组，这可能会对使用非英语的国际化程序产生不正确的结果。前面语句的结果可以使用 `Graphics` 方法 `drawBytes`，按照与 `drawChars` 输出一个字符数组相同的方式来输出一个字节数组。

## 8.5 比较 String

Java 提供了各种方法用以比较下面两个例子中给出的 `String` 对象。为了理解一个字符串“大于”或“小于”另一个字符串，我们考虑按字母排序姓氏的过程。读者会毫不犹豫地将“Jones”放在“Smith”之前，因为在字母表中“Jones”的首字母比“Smith”的首字母靠前。但字母表不仅仅是 26 个字母的列表，它同时也是字符的有序列表，每个字母都位于该表中的一个特定位置上。“Z”不仅仅是字母表中的一个字母，更是字母表中的第 26 个字母。

计算机是如何知道一个字母在另一个字母之前呢？所有的字符在计算机中都表示成数字编码；当计算机比较两个字符串时，它实际上是比较字符串中字符的数字编码。

图 8.3 显示了 `String` 方法 `equals`、`equalsIgnoreCase`、`compareTo` 以及 `regionMatches`，同时演示了如何使用相等运算符（`==`）来比较 `String` 对象。

```
1 // Fig. 8.3: StringCompare
2 // This program demonstrates the methods equals,
3 // equalsIgnoreCase, compareTo, and regionMatches
4 // of the String class.
5 import java.awt.Graphics;
6 import java.applet.Applet;
7
8 public class StringCompare extends Applet {
9     String s1, s2, s3, s4;
```

```
10
11     public void init()
12     {
13         s1 = new String( "hello" );
14         s2 = new String( "good bye" );
15         s3 = new String( "Happy Birthday" );
16         s4 = new String( "happy birthday" );
17     }
18
19     public void paint( Graphics g )
20     {
21         g.drawString( "s1 = " + s1, 25, 25 );
22         g.drawString( "s2 = " + s2, 25, 40 );
23         g.drawString( "s3 = " + s3, 25, 55 );
24         g.drawString( "s4 = " + s4, 25, 70 );
25
26         // test for equality
27         if ( s1.equals( "hello" ) )
28             g.drawString( "s1 equals \ "hello \ ", 25, 100 );
29         else
30             g.drawString( "s1 does not equal \ "hello \ ",
31                           25, 100 );
32
33         // test for equality with ==
34         if ( s1 == "hello" )
35             g.drawString( "s1 equals \ "hello \ ", 25, 115 );
36         else
37             g.drawString( "s1 does not equal \ "hello \ ",
38                           25, 115 );
39
40         // test for equality--ignore case
41         if ( s3.equalsIgnoreCase( s4 ) )
42             g.drawString( "s3 equals s4", 25, 130 );
43         else
44             g.drawString( "s3 does not equal s4", 25, 130 );
45
46         // test compareTo
47         g.drawString( "s1.compareTo( s2 ) is " +
48                       s1.compareTo( s2 ), 25, 160 );
49         g.drawString( "s2.compareTo( s1 ) is " +
50                       s2.compareTo( s1 ), 25, 175 );
51         g.drawString( "s1.compareTo( s1 ) is " +
52                       s1.compareTo( s1 ), 25, 190 );
53         g.drawString( "s3.compareTo( s4 ) is " +
54                       s3.compareTo( s4 ), 25, 205 );
55         g.drawString( "s4.compareTo( s3 ) is " +
56                       s4.compareTo( s3 ), 25, 220 );
57
58         // test regionMatches (case sensitive)
59         if ( s3.regionMatches( 0, s4, 0, 5 ) )
60             g.drawString(
61                 "First 5 characters of s3 and s4 match",
62                 25, 250 );
63         else
64             g.drawString(
65                 "First 5 characters of s3 and s4 do not match",
66                 25, 250 );
67     }
```

```

68         // test regionMatches (ignore case)
69         if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
70             g.drawString(
71                 "First 5 characters of s3 and s4 match",
72                 25, 265 );
73         else
74             g.drawString(
75                 "First 5 characters of s3 and s4 do not match",
76                 25, 265 );
77     }
78 }

```

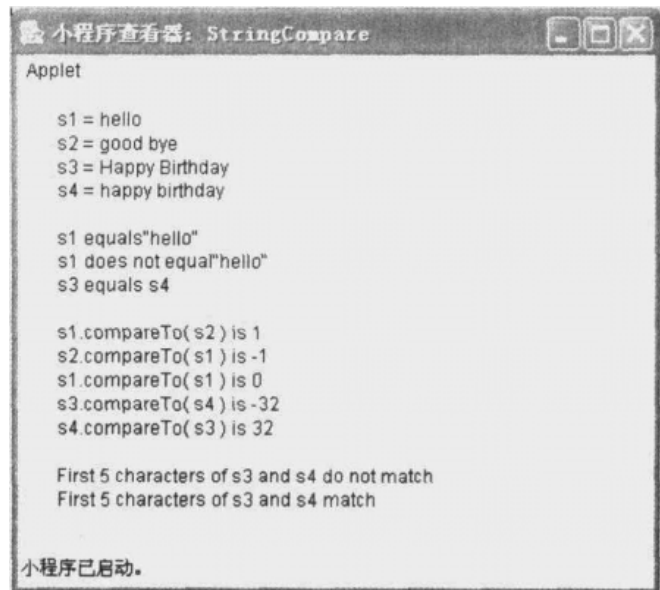


图 8.3 比较 String 对象

第 27 行的 if 结构的条件:

```
s1.equals ( "hello")
```

使用了 equals 方法来比较字符串 s1 和匿名字符串 “hello” 是否相等。equals 方法（由 String 类从其超类 Object 继承而来）用于测试任意两个对象的相等性（即两个对象内容是否相等），如果相等则返回 true，否则返回 false。本例中对 equals 调用的结果是 true，因为字符串 s1 已由匿名字符串 “hello” 的一份副本进行初始化了。equals 方法使用词典比较法（lexicographical comparison），即比较每个字符串中的各个字符在 Unicode 中的整数值。因此，如果将字符串 “hello” 同字符串 “HELLO” 进行比较，则结果为 false，因为小写字母的整数表示不同于其相应的大写字母。

第 34 行的 if 结构的条件:

```
s1=="hello"
```

使用相等运算符（==）来比较字符串 s1 和匿名字符串 “hello” 的相等性。运算符 “==” 在比较引用和比较基本数据类型时具有不同的功能。当基本数据类型使用 “==” 进行比较时，如果两个值相同，结果就是 true。当对引用进行 “==” 比较时，如果两个引用都指向内存中的同一个对象，则结果为 true。为了比较对象实际内容（或状态信息）的相等性，必须调用特定的方法（如 equals）。前面的条件判断为 false，因为引用 s1 是使用下列语句初始化的：



```
s1 =new String ("hello");
```

该语句使用匿名字符串“hello”的副本创建了一个新的String对象，并将新对象赋给了引用s1。如果s1是使用下列语句初始化的：

```
s1="hello"
```

则该语句直接将匿名字符串“hello”赋给引用s1，这样条件判断产生的结果为true。这是因为Java将相同内容的所有匿名String对象作为有许多引用的同一个匿名String对象。因此，第13行、第27行和第34行都指向内存中的同一个匿名字符串对象“hello”。

如果对字符串排序，那么可以使用equalsIgnorecase方法来比较其相等性，该方法忽略了比较过程中每个字符串字母的大小写。因此，字符串“hello”和字符串“HELLO”是相等的。第41行的if结构使用下列条件：

```
s3.equalsIgnoreCase(s4);
```

来比较字符串s3（Happy Birthday）与字符串s4（happy birthday）的相等性，比较的结果为true。

第47行到第56行使用String方法CompareTo来比较String对象。例如，下列表达式：

```
s1.compareTo(s2)
```

在第48行上比较字符串s1和字符串s2。如果这两个字符串是相等的，则compareTo方法返回0；如果激活compareTo方法的字符串小于作为参数传递的字符串，则返回一个负数；如果激活compareTo方法的字符串大于作为参数传递的字符串，则返回一个正数。compareTo方法使用词典比较法，返回值为两个String中第一个不同字符的整数表示的差值。当比较s4和s3时，两个字符串中的第一个字符就不相同，因此返回值为32，因为h的整数表示为104而H的整数表示为72。

第59行上if结构的条件：

```
s3.regionMatches ( 0, s4 , 0 , 5)
```

使用String方法regionMathes来比较两个String对象相同的一部分。第一个参数是激活该方法的String的开始下标，第二个参数是进行比较的另一个String，第三个参数是被比较的String的开始下标，最后一个参数是在两个String之间进行比较的字符个数。仅当指定数目的字符的词典序数相等时，该方法才返回true。

最后，第69行的if结构的条件：

```
s3.regionMathes( true , 0 , s4 , 0 , 5 )
```

使用String方法regionMathes的第二个版本来比较两个String对象的相同部分。如果第一个参数为true，则该方法就不区分所比较字符的大小写。其余的参数与带有4个参数的regionMathes方法中描述的相同。

本节的第二个例子（如图8.4所示）演示了String类的startsWith方法和endsWith方法。applet StringStartEnd包含了一个String数组，称为string，后者又包含了“started”、“starting”、“ended”以及“ending”。paint方法包含了3个for结构，用来测试数组的元素，判断它们是否以一个特定的字符集开始或结束。

```
1 // Fig. 8.4: StringStartEnd.java
2 // This program demonstrates the methods startsWith and
3 // endsWith of the String class.
```

```
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class StringStartEnd extends Applet{
8     String strings[] = { "started", "starting",
9                           "ended", "ending" };
10
11     public void paint( Graphics g )
12     {
13         int yPosition = 25;
14
15         // Test method startsWith
16         for ( int i = 0; i < strings.length; i++ )
17             if ( strings[ i ].startsWith( "st" ) ) {
18                 g.drawString( " \ " + strings[ i ] +
19                             " \ " starts with \ "st \ ", 25, yPosition );
20                 yPosition += 15;
21             }
22
23         yPosition += 15;
24
25         // Test method startsWith starting from position
26         // 2 of the string
27         for ( int i = 0; i < strings.length; i++ )
28             if ( strings[ i ].startsWith( "art", 2 ) ) {
29                 g.drawString( " \ " + strings[ i ] +
30                             " \ " starts with \ "art \ " at position 2",
31                             25, yPosition );
32                 yPosition += 15;
33             }
34
35         yPosition += 15;
36
37         // Test method endsWith
38         for ( int i = 0; i < strings.length; i++ )
39             if ( strings[ i ].endsWith( "ed" ) ) {
40                 g.drawString( " \ " + strings[ i ] +
41                             " \ " ends with \ "ed \ ", 25, yPosition );
42                 yPosition += 15;
43             }
44     }
45 }
```

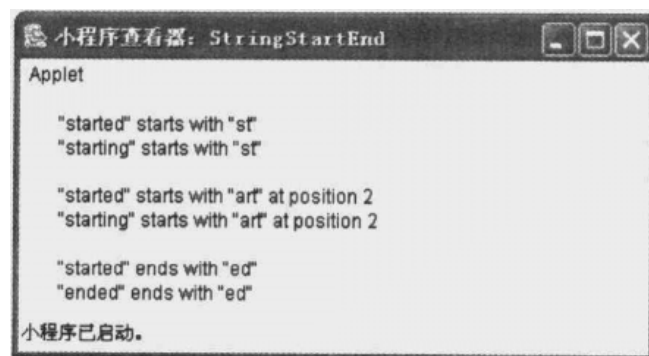


图 8.4 String 类的 startsWith 和 endsWith 方法

第一个 for 结构 (第 16 行) 使用以一个 String 为参数的 `startsWith` 方法。if 结构 (第 17 行) 的条件:

```
strings [ i ].startsWith ( "st" )
```

判断数组 `strings` 的位置 `i` 上的 String 是否以字符串 “st” 开始。如果是, 则方法返回 `true`, 并且在 applet 上显示 String。

第二个 for 结构 (第 27 行) 使用以一个 String 和一个整数为参数的 `startsWith` 方法。整型参数说明了应在该 String 中开始比较的下标。if 结构的条件:

```
strings [ i ].startsWith ( "art" , 2 )
```

判断数组 `strings` 位置 `i` 上的 String 在下标 2 上的字符是否以 “art” 开始。如果是, 该方法就返回 `true`, 并且在 applet 上显示该 String。

第三个 for 结构 (第 38 行) 使用 `endsWith`, 它以一个 String 为参数。if 结构的条件:

```
strings [ i ].endsWith ( "ed" )
```

判断数组 `strings` 的位置 `i` 上的 String 是否以字符串 “ed” 结尾。如果是, 则该方法就返回 `true`, 并且在 applet 上显示该 String。

## 8.6 String 方法 hashCode

我们经常需要以一种能快速找到信息的方式来存储 String 和其他的数据类型。最好的办法之一就是将信息存储在一个散列 (hash) 表中。一个散列表利用散列代码 (hash code) 来存储信息。散列代码是对要存储的对象进行特殊计算而产生的。散列代码用于选择对象在表中的存储位置。当需要取出信息时, 又将执行相同的计算, 首先确定散列代码, 通过对表中特定位置的查找来得到存储在该位置的值。每个对象都可以存储在一个散列表中。Object 类定义了 `hashCode` 方法来完成散列代码的计算。Object 的所有子类中都将继承该方法。String 重写了 `hashCode` 方法, 以便提供一个基于该 String 内容的良好散列代码分布。我们将在第 18 章详细地讨论散列表。

图 8.5 中的例子展示了包含 “hello” 和 “Hello” 的两个 String 的 `hashCode` 方法。

```
1 // Fig. 8.5: StringHashCode.java
2 // This program demonstrates the method
3 // hashCode of the String class.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class StringHashCode extends Applet {
8     String s1 = "hello",
9         s2 = "Hello";
10
11     public void paint( Graphics g )
12     {
13         g.drawString( "The hash code for \ " +
14             s1 + " \ " is " + s1.hashCode(), 25, 25 );
15         g.drawString( "The hash code for \ " +
16             s2 + " \ " is " + s2.hashCode(), 25, 40 );
17     }
18 }
```



图 8.5 String 类的 hashCode 方法

注意，每个 String 的散列代码值都是不同的，因为 String 本身在词典顺序上是不同的，我们将在第 18 章详细地讨论散列表。

## 8.7 在 String 中定位字符和子字符串

在 String 中搜寻一个字符或字符集是很常见的。例如，如果要创建自己的字符串处理器，那么可能要提供在整个文档中进行查找的功能。图 8.6 中的程序展示了 String 方法 indexOf 和 lastIndexOf 的诸多版本，这些方法在 String 中查找一个特定的字符。本例的所有查找都是在 applet StringIndexMethods 的 paint 方法的 String letters（初始化成“abcdefghijklmabcdefghijklm”）上完成的。

```
1 // Fig. 8.6: StringIndexMethods.java
2 // This program demonstrates the String
3 // class index methods.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class StringIndexMethods extends Applet {
8     String letters = "abcdefghijklmabcdefghijklm";
9
10    public void paint( Graphics g )
11    {
12        // test indexOf to locate a character in a string
13        g.drawString( "'c' is located at index " +
14            letters.indexOf( (int) 'c' ), 25, 25 );
15
16        g.drawString( "'a' is located at index " +
17            letters.indexOf( (int) 'a', 1 ), 25, 40 );
18
19        g.drawString( "'$' is located at index " +
20            letters.indexOf( (int) '$' ), 25, 55 );
21
22        // test lastIndexOf to find a character in a string
23        g.drawString( "Last 'c' is located at index " +
24            letters.lastIndexOf( (int) 'c' ), 25, 85 );
25
26        g.drawString( "Last 'a' is located at index " +
27            letters.lastIndexOf( (int) 'a', 25 ), 25, 100 );
28
29        g.drawString( "Last '$' is located at index " +
30            letters.lastIndexOf( (int) '$' ), 25, 115 );
31    }
```

```

32      // test indexOf to locate a substring in a string
33      g.drawString( " \ "def \ " is located at index " +
34                  letters.indexOf( "def" ), 25, 145 );
35
36      g.drawString( " \ "def \ " is located at index " +
37                  letters.indexOf( "def", 7 ), 25, 160 );
38
39      g.drawString( " \ "hello \ " is located at index " +
40                  letters.indexOf( "hello" ), 25, 175 );
41
42      // test lastIndexOf to find a substring in a string
43      g.drawString( "Last \ "def \ " is located at index " +
44                  letters.lastIndexOf( "def" ), 25, 205 );
45
46      g.drawString( "Last \ "def \ " is located at index " +
47                  letters.lastIndexOf( "def", 25 ), 25, 220 );
48
49      g.drawString( "Last \ "hello \ " is located at index " +
50                  letters.lastIndexOf( "hello" ), 25, 235 );
51  }
52  }

```

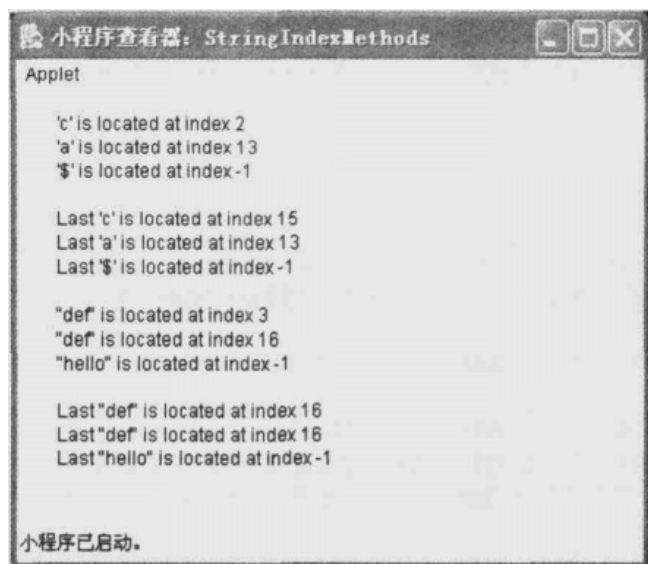


图 8.6 String 类的查找方法

第13行~第20行的输出语句使用indexOf方法来定位String中某个字符首次出现的位置。如果发现了该字符，则返回String中该字符的下标，否则返回-1。这里有两条用于在String中查找字符的indexOf语句。下列表达式：

```
letters.indexOf( (int) 'c' )
```

使用以一个整数为参数的indexOf方法，该参数是一个字符的整数表示，这也是上述表达式中进行整型转换的原因。下列表达式：

```
letters.indexOf( (int) 'a' , 1 )
```

使用indexOf方法的第二个版本，该方法以两个整数为参数，参数为一个字符的整数表示和对String进行查找的开始下标。

第 23 行 ~ 第 30 行的输出语句使用 `lastIndexOf` 方法来定位 `String` 中某个字符最后一次出现的位置, 从 `String` 结尾向 `String` 开头进行查找。如果找到该字符, 则返回字符串中该字符的下标; 否则返回 `-1`。这里有两处使用 `lastIndexOf` 方法在 `String` 中查找字符的语句。下列表达式:

```
letters.indexOf( (int) 'c' )
```

使用以一个整数为参数的 `lastIndexOf` 方法, 该参数是一个字符的整数表示。下列表达式:

```
letters.lastIndexOf( (int) 'a' , 25 )
```

使用以两个整数为参数的 `lastIndexOf` 方法, 参数为一个字符的整数表示和开始查找的起始下标。

第 33 行 ~ 第 50 行的程序展示了 `indexOf` 和 `lastIndexOf` 方法, 它们分别以一个 `String` 为其第一个参数。这些方法的不同版本与上面描述的完全相同, 除了它们查找由其 `String` 参数指定的字符集(或子字符串)以外。

## 8.8 从 `String` 中提取子字符串

`String` 类提供了两个 `substring` 方法, 通过复制已有 `String` 对象的一部分来创建新的 `String` 对象。这两个方法都返回一个 `String` 对象, 如图 8.7 中所示。

```

1 // Fig. 8.7: SubString.java
2 // This program demonstrates the String class substring methods.
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class SubString extends Applet {
7     String letters = "abcdefghijklmabcdefghijklm";
8
9     public void paint( Graphics g )
10    {
11        // test substring methods
12        g.drawString( "Substring from index 20 to end is " +
13            " \ " + letters.substring( 20 ) + " \ ", 25, 25 );
14
15        g.drawString( "Substring from index 0 upto 6 is " +
16            " \ " + letters.substring( 0, 6 ) + " \ ", 25, 40 );
17    }
18 }
```

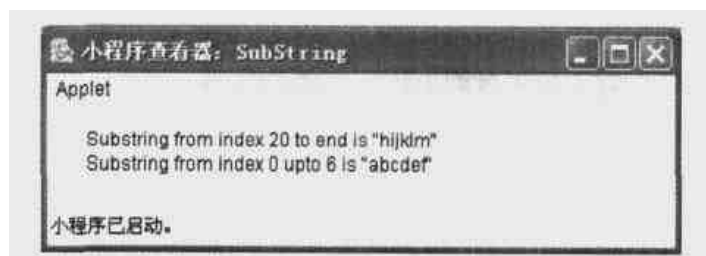


图 8.7 `String` 类的 `substring` 方法

下列表达式:

```
letters.substring( 20 )
```

使用以一个整数为参数的 `substring` 方法, 该参数指明在原始 `String` 中要复制的字符的开始下标。返回的子字符串包含从 `String` 的开始下标到结尾的字符。如果将下标指定为超过 `String` 的边界之外的参数, 则会生成 `StringIndexOutOfBoundsException` 异常。

下列表达式:

```
letters.substring ( 0, 6 )
```

使用以两个整数为参数的 `substring` 方法。第一个参数指定了在原始 `String` 中要复制的第一个字符的下标, 第二个参数指定了要复制的最后一个字符之后的位置下标。返回的子字符串包含来自于原始 `String` 的指定字符的副本。如果指定的参数超出了 `String` 的边界, 则将生成 `StringIndexOutOfBoundsException` 异常。

## 8.9 连接 String

使用 `String` 的 `concat` 方法 (如图 8.8 所示) 可以连接两个 `String` 对象, 并返回一个新的 `String` 对象, 这个新对象包含来自于两个原始 `String` 的字符。如果参数 `String` 不包含字符, 则返回原始的 `String`。

```
1 // Fig. 8.8: StringConcat.java
2 // This program demonstrates the String class concat method.
3 // Note that the concat method returns a new String object. It
4 // does not modify the object that invoked the concat method.
5 import java.awt.Graphics;
6 import java.applet.Applet;
7
8 public class StringConcat extends Applet {
9     String s1 = new String( "Happy " ),
10     s2 = new String( "Birthday" );
11
12     public void paint( Graphics g )
13     {
14         g.drawString( "s1 = " + s1, 25, 25 );
15         g.drawString( "s2 = " + s2, 25, 40 );
16         g.drawString( "Result of s1.concat( s2 ) = " +
17                     s1.concat( s2 ), 25, 55 );
18         g.drawString( "s1 after concatenation = " + s1,
19                     25, 70 );
20     }
21 }
```

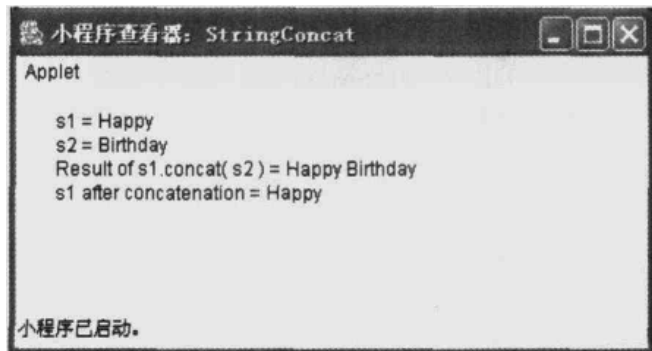


图 8.8 `String` 类的 `concat` 方法

下列表达式:

```
s1.concat ( s2 )
```

向 String s1 的末尾添加 String s2 的字符, 并且不修改原始的 String s1 和 s2。

## 8.10 其他的 String 方法

String 类提供了几个方法, 可以返回更改的 String 副本或一个字符数组。在 applet StringMisc2 (如图 8.9 所示) 的 paint 方法中展示了这些方法。

下列表达式:

```
s1.replace('l' , 'L' )
```

在第 21 行上使用 String 方法 replace 来生成一个新的 String 对象。在 String s1 中每次出现字符 'l' 时, 将使用字符 'L' 进行替换。该方法返回一个新的包含已替换 String 的 String 对象; 如果第一个参数没有在 String 中出现, 那么就返回原始的 String。这里没有更改 String 对象。

```
1 // Fig. 8.9: StringMisc2.java
2 // This program demonstrates the String class replace,
3 // toLowerCase, toUpperCase, trim, toString, and toCharArray
4 // methods.
5 import java.awt.Graphics;
6 import java.applet.Applet;
7
8 public class StringMisc2 extends Applet {
9     String s1 = new String( "hello" ),
10         s2 = new String( "GOOD BYE" ),
11         s3 = new String( "spaces" );
12
13     public void paint( Graphics g )
14     {
15         g.drawString( "s1 = " + s1, 25, 25 );
16         g.drawString( "s2 = " + s2, 25, 40 );
17         g.drawString( "s3 = " + s3, 25, 55 );
18
19         // test method replace
20         g.drawString( "Replace 'l' with 'L' in s1: " +
21             s1.replace( 'l', 'L' ), 25, 85 );
22
23         // test toLowerCase and toUpperCase
24         g.drawString( "s1 after toUpperCase = " +
25             s1.toUpperCase(), 25, 115 );
26         g.drawString( "s2 after toLowerCase = " +
27             s2.toLowerCase(), 25, 130 );
28
29         // test trim method
30         g.drawString( "s3 after trim = \" " + s3.trim() + " \",
31             25, 160 );
32
33         // test toString method
34         g.drawString( "s1 = " + s1.toString(), 25, 190 );
35     }
}
```



```

36      // test toCharArray method
37      char charArray[] = s1.toCharArray();
38      g.drawString( "s1 as a character array = ", 25, 220 );
39      g.drawChars( charArray, 0, charArray.length, 172, 220 );
40  }
41  }

```

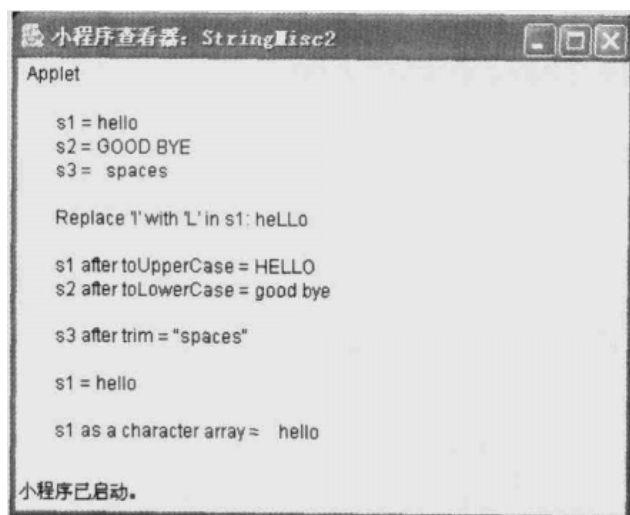


图 8.9 其他的 String 方法

下列表达式:

```
s1.toUpperCase( )
```

在第 25 行上使用 String 方法 toUpperCase 来生成一个新 String 对象, 并用大写字母替换 s1 中相应的小写字母。该方法返回一个包含了转换过的 String 的新 String; 如果在 String 中的相应字符没有大写版本, 则返回原始的 String。原始的 String 并没有改动。

下列表达式:

```
s2.toLowerCase( )
```

在第 27 行利用 String 方法 toLowerCase 来生成一个新 String 对象, 该方法使用小写字母替换 s2 中相应的大写字母。该方法返回一个新的 String 对象, 它包含转换过的 String; 如果 String 中的字符没有小写版本, 则返回原始的 String。原始的 String 并没有改动。

下列表达式:

```
s3.trim( )
```

在第 30 行利用 String 方法 trim 生成一个新的 String 对象, 它删去了 String 的头部和尾部出现的所有空白字符 (如空格、换行符和制表符)。该方法返回一个新的 String 对象, 它包含无前导和后续空白字符的 String。原始的 String 并没有改动。

下列表达式:

```
s1.toString( )
```

在第 34 行返回 String s1。为什么要为 String 类提供 toString 方法呢? 在 Java 中, 所有的对象都能利用方法 toString 而转换成 String, toString 源自于 Object 类。如果一个类继承自 Object (如 String) 而

没有提供重写的方法 `toString`，则使用 `Object` 类的默认版本。默认版本创建了一个 `String`，它包含该对象的类名和对象的散列代码。`toString` 方法一般用于使用文字表达一个对象的内容。在 `String` 类中提供 `toString` 方法，是为了保证返回合适的 `String` 值。

下列语句：

```
charcharArray[ ] = s1.toCharArray ( );
```

在 37 行创建了一个新的字符数组，其中包含 `String s1` 中字符的一份副本。并将其赋给 `charArray`。

## 8.11 使用 `String` 方法 `valueOf`

`String` 类提供了一组以各种类型为参数的 `static` 类方法，这些方法将这些参数转换成字符串，并将它们作为 `String` 对象返回。`StringValueOf` 类（如图 8.10 所示）展示了 `String` 类的 `valueOf` 方法。

`String.valueOf` 表达式 (`charArray`) 在第 19 行把字符数组 `charArray` 的内容复制到一个新的 `String` 对象并返回这个 `String`。

```

1 // Fig. 8.10: StringValueOf.java
2 // This program demonstrates the String class valueOf methods.
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class StringValueOf extends Applet {
7     char charArray[ ] = { 'a', 'b', 'c', 'd', 'e', 'f' };
8     boolean b = true;
9     char c = 'Z';
10    int i = 7;
11    long l = 10000000;
12    float f = 2.5f;
13    double d = 33.333;
14    Object o = "hello"; // Assign String to Object reference
15
16    public void paint( Graphics g )
17    {
18        g.drawString( "char array = " +
19                      String.valueOf( charArray ), 25, 25 );
20        g.drawString( "part of char array = " +
21                      String.valueOf( charArray, 3, 3 ),
22                      25, 40 );
23
24        g.drawString( "boolean = " + String.valueOf( b ),
25                      25, 70 );
26        g.drawString( "char = " + String.valueOf( c ),
27                      25, 85 );
28        g.drawString( "int = " + String.valueOf( i ),
29                      25, 100 );
30        g.drawString( "long = " + String.valueOf( l ),
31                      25, 115 );
32        g.drawString( "float = " + String.valueOf( f ),
33                      25, 130 );
34        g.drawString( "double = " + String.valueOf( d ),
35                      25, 145 );
36        g.drawString( "Object = " + String.valueOf( o ),

```

```

37         25, 160 );
38     }
39 }

```

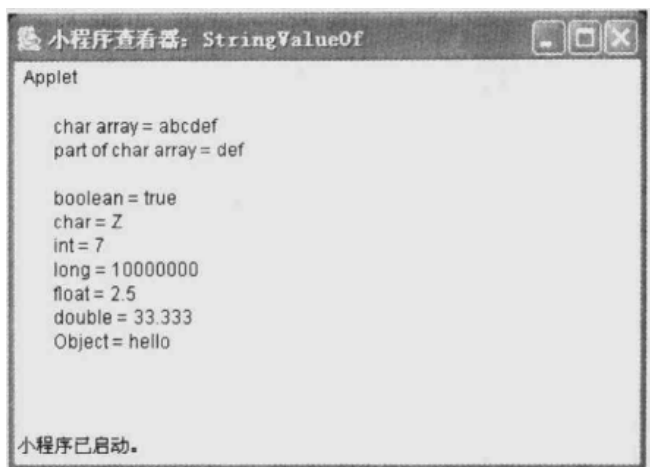


图 8.10 String 类的 valueOf 方法

下列表达式:

```
String.valueOf ( charArray , 3 , 3 )
```

在第21行把字符数组 charArray 的部分内容复制到一个新的 String 对象并返回这个 String。第二个参数指明了要复制的字符的开始下标，第三个参数指明了要复制的字符个数。

valueOf 方法还有其他 7 个版本，分别以 boolean、char、int、long、float、double 和 object 类型为参数，在程序的第24行~第37行给出了这些类型。注意，以 Object 为参数的 valueOf 方法之所以可行，是因为可以使用 toString 方法将 Object 对象转换成 String 对象。

## 8.12 String 方法 intern

比较大型的 String 对象是一个相对较慢的操作，String 方法 intern 可以改进 String 的比较性能。String 方法 intern 由一个 String 对象激活时，它将返回对一个 String 对象的引用（该对象与原始 String 含有相同的内容）。String 类在程序执行过程中保留得到的 String 引用。这样，如果程序在其他 String 对象上调用 intern 方法，且该对象与原始 String 对象含有相同的内容，那么该方法将返回内存中保留的 String 副本的引用。这对大型 String 的高效比较是有用的。执行 intern 后，String 引用就可以通过“==”（即比较两个引用，这是一种快速操作）来比较，而并非使用诸如 equals 和 equalsIgnoreCase 之类的比较方法（后者需要在每个 String 中比较一对字符，是一种耗时的循环操作）。图 8.11 的程序展示了 intern 方法。

applet 类 StringIntern 声明了 4 个 String: s1, s2, s3, s4。String s1 和 s2 使用“hello”的副本进行初始化。第一个 if 结构使用运算符“==”来判断 String s1 和 s2 在内存中是否为同一个对象。第二个 if 结构使用方法 equals 来判断 String s1 和 s2 的内容是否相等。下列语句:

```
s3 = s1.intern ( );
```

使用 intern 方法来获取对一个 String 对象（与 s1 对象含有相同内容的）的引用，并将引用赋给 s3。s3 引用（即 s1.intern() 返回的对象的引用而非 s1）的 String 仍由 String 类保留在内存中。下列语句:



```
51         else
52             g.drawString( "s1 and s3 are not the " +
53                           "same object in memory", 25, 85 );
54
55         // Determine if s2 and s4 refer to the same object
56         if ( s2 == s4 )
57             g.drawString( "s2 and s4 are the " +
58                           "same object in memory", 25, 105 );
59         else
60             g.drawString( "s2 and s4 are not the " +
61                           "same object in memory", 25, 105 );
62
63         // Determine if s1 and s4 refer to the same object
64         if ( s1 == s4 )
65             g.drawString( "s1 and s4 are the " +
66                           "same object in memory", 25, 125 );
67         else
68             g.drawString( "s1 and s4 are not the " +
69                           "same object in memory", 25, 125 );
70     }
71 }
```

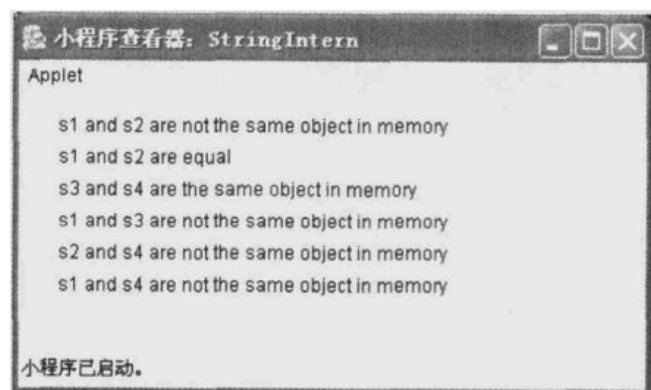


图 8.11 String 类的 intern 方法

第三个if结构使用运算符“==”来确定String s3和s4在内存中是相同的对象。第四个if结构使用运算符“==”来确定String s1和s3不是内存中的相同对象。第五个if结构使用运算符“==”来确定String s2和s4不是内存中的相同对象，这是因为第二个intern调用导致了对s1.intern()返回的对象的引用，而非String s2的引用。第六个if结构使用运算符“==”来确定String s1和s4实际上不是内存中的相同对象，因为第二个intern调用导致了对s1.intern()返回的对象的引用，而非String s1。

### 8.13 StringBuffer 类

String类提供了许多处理String的功能。然而，一旦创建了String对象以后，它的内容就不再改变。下而几节要讨论用于创建和操作动态字符串的内容，即可修改String的StringBuffer类的特性。每个StringBuffer都能够存储由其性能规定的字符数目，如果超出了—个StringBuffer的容量，则其容量便会自动扩展以容纳新增的字符。正如我们将要看到的，StringBuffer类也用于实现运算符“+”和“+=”，它们用于String的连接。

### 性能提示 8.1

String 对象是常量字符串，而 StringBuffer 对象则是可修改字符串。Java 为优化的目的而区分常量字符串与可修改字符串；特别是 Java 可以完成涉及 String 对象的特定优化操作（如多重引用共享一个 String 对象），因为它知道这些对象将不再改变。

### 性能提示 8.2

在确定对象不会改变的情况下，选择是使用 String 对象代表一个字符串，还是使用 StringBuffer 对象代表这个字符串时，最好使用 String 对象，这样可以改进性能。

### 常见编程错误 8.3

对 String 对象使用只在 StringBuffer 中才有而在 String 中没有的方法将会产生语法错误。

## 8.14 StringBuffer 构造函数

StringBuffer 类提供了的三个构造函数，如图 8.12 所示。

```
1 // Fig. 8.12: StringBufferConstructors.java
2 // This program demonstrates the StringBuffer constructors.
3 import java.awt.Graphics;
4 import java.applet.Applet;
5
6 public class StringBufferConstructors extends Applet {
7     StringBuffer buf1, buf2, buf3;
8
9     public void init()
10    {
11        buf1 = new StringBuffer();
12        buf2 = new StringBuffer( 10 );
13        buf3 = new StringBuffer( "hello" );
14    }
15
16    public void paint( Graphics g )
17    {
18        g.drawString( "buf1 = " + " \ " + buf1.toString() + " \ ",
19                      25, 25 );
20        g.drawString( "buf2 = " + " \ " + buf2.toString() + " \ ",
21                      25, 40 );
22        g.drawString( "buf3 = " + " \ " + buf3.toString() + " \ ",
23                      25, 55 );
24    }
25 }
```

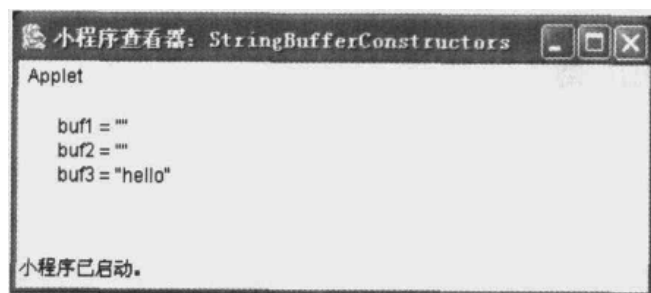


图 8.12 String 构造函数

下列语句:

```
buf1 = new StringBuffer ( );
```

使用默认 `StringBuffer` 构造函数创建了一个没有字符的 `StringBuffer`, 其最初的容量为 16 个字符。

下列语句:

```
buf2 = new StringBuffer ( 10 );
```

使用以一个整数为参数的构造函数, 创建了一个没有字符的 `StringBuffer`, 其最初的容量由整数参数规定。

下列语句:

```
buf3 = new StringBuffer ( "hello" );
```

使用以一个 `String` 为参数的构造函数, 创建了一个包含 `String` 参数中字符的 `StringBuffer`, 最初的容量是 `String` 参数中的字符数加上 16。

第 18 行 ~ 第 23 行的输出语句使用了 `StringBuffer` 的 `toString` 方法, 将 3 个 `StringBuffer` 转换成 `String` 对象, 从而可以使用 `drawString` 方法显示出来。注意, 使用了运算符 “+” 来连接输出的 `String`。在 8.17 节中, 我们将讨论 `StringBuffer` 是如何用于实现 运算符 “+” 和 “+=” 的。

## 8.15 `StringBuffer` 的 `length`、`capacity`、`setLength` 和 `ensureCapacity` 方法

`StringBuffer` 类提供了 `length` 和 `capacity` 方法, 分别返回一个 `StringBuffer` 中当前的字符数, 以及在分配更多内存的情况下一个 `StringBuffer` 所能存储的字符个数。`ensureCapacity` 方法向程序员提供了保证 `StringBuffer` 最小容量的方式。`setLength` 方法使程序员可以增加或减小一个 `StringBuffer` 的长度, 如图 8.13 所示。

```
1 // Fig. 8.13: StringBufferCapLen.java
2 // This program demonstrates the length and
3 // capacity methods of the StringBuffer class.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class StringBufferCapLen extends Applet {
8     StringBuffer buf;
9
10    public void init()
11    {
12        buf = new StringBuffer( "hello" );
13    }
14
15    public void paint ( Graphics g )
16    {
17        g.drawString( "buf = " + buf.toString(), 25, 25 );
18        g.drawString( "length = " + buf.length(), 25, 40 );
19        g.drawString( "capacity = " + buf.capacity(),
20                    25, 55 );
21    }
```

```
22         buf.ensureCapacity( 50 );
23         g.drawString( "New capacity = " + buf.capacity(),
24                       25, 85 );
25
26         buf.setLength( 10 );
27         g.drawString( "New length = " + buf.length(),
28                       25, 115 );
29     }
30 }
```

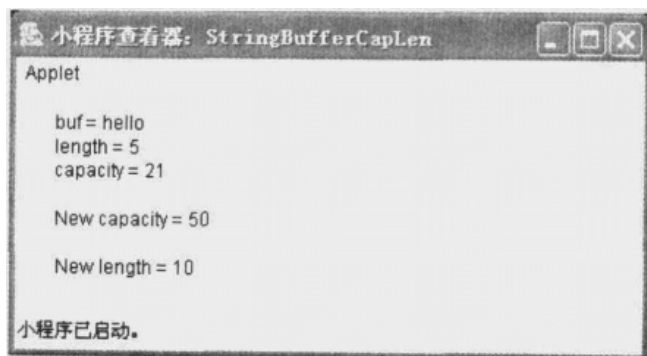


图 8.13 StringBuffer 类的 length 和 capacity 方法

applet `StringBufferCapLen` 类包含了一个实例变量——`StringBuffer` 对象 `buf`。程序的第 12 行使用 `StringBuffer` 的以一个 `String` 为参数的构造函数来进行实例化，并使用“hello”初始化该 `StringBuffer`。注意，在输出窗口中将 `StringBuffer` 的容量初始化为 21。记住，以一个 `String` 为参数的 `StringBuffer` 构造函数创建了一个 `StringBuffer` 对象，其容量为传递而来的 `String` 参数的长度加上 16。

下列语句：

```
buf.ensureCapacity( 50 );
```

扩展了 `StringBuffer` 的容量为最小 50 个字符。实际上，如果原始的容量小于此参数，则该方法保证容量大于参数规定的数目，或者为原始容量的两倍再加 2。如果 `StringBuffer` 的当前容量大于指定的容量，则 `StringBuffer` 的容量保持不变。

下列语句：

```
buf.setLength(10);
```

使用 `setLength` 方法来设置 `StringBuffer` 的长度为 10。如果规定的长度小于 `StringBuffer` 中当前的字符个数，则将字符截尾到规定的长度——即 `StringBuffer` 中在规定长度之后的字符都将丢弃。如果规定的长度大于 `StringBuffer` 中当前的字符个数，则将空白字符（数字表示为 0 的字符）附加到 `StringBuffer` 之后，直到 `StringBuffer` 中的字符数等于规定的长度为止。

## 8.16 StringBuffer 的 charAt、setCharAt 和 getChars 方法

`StringBuffer` 类提供了 `charAt`、`setCharAt` 和 `getChars` 方法，用以操作一个 `StringBuffer` 中的字符。`CharAt` 方法接收一个整型参数，并返回 `StringBuffer` 中该下标处的字符。`setCharAt` 方法接收一个整数和一个字符为参数，从而将字符置于指定的位置上。`charAt` 和 `setCharAt` 方法中的下标必须大于等于 0 且小于 `StringBuffer` 的长度，否则将产生一个 `StringIndexOutOfBoundsException` 异常。



## 常见编程错误 8.4

试图访问 StringBuffer 边界之外的一个字符 (即小于 0 或大于等于 StringBuffer 长度的下标), 将导致一个 `StringIndexOutOfBoundsException` 异常。

`getChars` 方法返回包含 StringBuffer 中字符副本的一个字符数组。该方法接收 4 个参数: StringBuffer 中应复制字符的起始下标, StringBuffer 中应复制字符之后的那个位置的下标, 字符将被复制到字符数组, 以及字符数组中放置首字符的位置。每个方法都在图 8.14 中展示出来。

```
1 // Fig. 8.14: StringBufferChars.java
2 // The charAt, setCharAt, and getChars methods
3 // of class StringBuffer.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class StringBufferChars extends Applet {
8     StringBuffer buf;
9     char charArray [ ];
10
11     public void init()
12     {
13         buf = new StringBuffer( "hello there" );
14         charArray = new char [ 100 ];
15     }
16
17     public void paint ( Graphics g )
18     {
19         g.drawString( "buf = " + buf.toString(), 25, 25 );
20         g.drawString( "Character at 0: " + buf.charAt( 0 ),
21                     25, 40 );
22         g.drawString( "Character at 4: " + buf.charAt( 4 ),
23                     25, 55 );
24
25         buf.getChars( 0, buf.length(), charArray, 0 );
26         g.drawString( "The characters are: ", 25, 85 );
27         g.drawChars( charArray, 0, charArray.length, 142, 85 );
28
29         buf.setCharAt( 0, 'H' );
30         buf.setCharAt( 6, 'T' );
31         g.drawString( "buf = " + buf.toString(), 25, 115 );
32     }
33 }
```

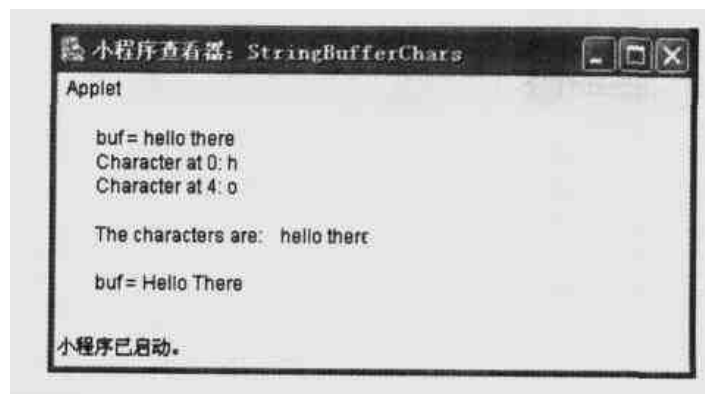


图 8.14 StringBuffer 类的字符操作方法

## 8.17 StringBuffer 的 append 方法

StringBuffer 类提供了 10 个重载的 append 方法, 允许将各种数据类型的值添加到一个 StringBuffer 的末尾。针对每个基本数据类型和字符数组, String 和 Object (记注, toString 方法可以产生任何 Object 的一个 String 表示) 都有对应版本。每个方法接收其参数, 将参数转换成一个 String 并附加到 StringBuffer 的末尾。append 方法在图 8.15 中展示。

```
1 // Fig. 8.15: StringBufferAppend.java
2 // This program demonstrates the append
3 // methods of the StringBuffer class.
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class StringBufferAppend extends Applet {
8     Object o = "hello"; // Assign String to Object reference
9     String s = "good bye";
10    char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
11    boolean b = true;
12    char c = 'Z';
13    int i = 7;
14    long l = 10000000;
15    float f = 2.5f;
16    double d = 33.333;
17    StringBuffer buf;
18
19    public void init()
20    {
21        buf = new StringBuffer();
22    }
23
24    public void start()
25    {
26        buf.append( o );
27        buf.append( ' ' );
28        buf.append( s );
29        buf.append( ' ' );
30        buf.append( charArray );
31        buf.append( ' ' );
32        buf.append( charArray, 0, 3 );
33        buf.append( ' ' );
34        buf.append( b );
35        buf.append( ' ' );
36        buf.append( c );
37        buf.append( ' ' );
38        buf.append( i );
39        buf.append( ' ' );
40        buf.append( l );
41        buf.append( ' ' );
42        buf.append( f );
43        buf.append( ' ' );
44        buf.append( d );
45    }
46}
```

```

47     public void paint( Graphics g )
48     {
49         g.drawString( "buf = " + buf.toString(), 25, 25 );
50     }
51 }

```

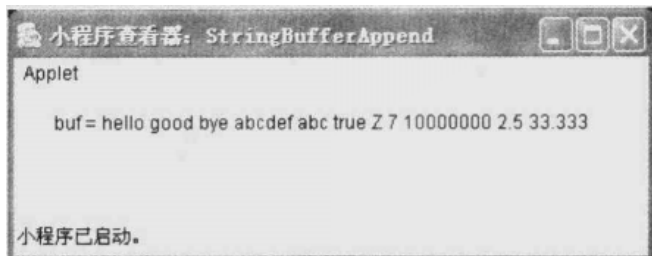


图 8.15 StringBuffer 类的 append 方法

实际上，编译器将 StringBuffer 和 append 方法用于实现连接 String 的 “+” 和 “+=” 运算符。例如，下列语句：

```
Strings = " BC " + 22;
```

连接 String “BC” 和整数 22，如下所示：

```
new StringBuffer ( ).append ( " BC" ).append (22).toString ( );
```

首先，创建一个空的 StringBuffer，接着将 String “BC” 附加到空的 StringBuffer 上。接下来将整数 22 附加到 StringBuffer 的末尾。最后，用 toString 方法将 StringBuffer 转换成一个 String，将结果赋给 String s。下列语句：

```
s+= "!";
```

实际上是按照以下形式执行的：

```
s = new StringBuffer ( ).append (s).append ("!").toString ( )
```

首先，创建一个空的 String Buffer，然后将 String s 附加到 StringBuffer 上。接着将 String “!” 附加到 StringBuffer 的末尾。最后，利用方法 toString 将 StringBuffer 转换成一个 String，结果赋给 String s。

## 8.18 StringBuffer 的 insert 方法

StringBuffer 类提供了 9 种重载的 insert 方法，允许各种数据类型插入到一个 StringBuffer 的任意位置上，针对每个原始数据类型，以及字符数组、String 和 Object（记住，toString 方法可用来生成任意对象的 String 表示）都提供了各种版本。每种方法都接收第二个参数，将其转换成一个 String 并插入到由第一个参数指定的下标位置；由第二个参数指定的下标必须大于等于 0，并且小于 StringBuffer 的长度，否则就会产生一个 StringIndexOutOfBoundsException 异常。在图 8.16 中展示了 insert 方法。

```

1    // Fig. 8.16: StringBufferInsert.java
2    // This program demonstrates the insert
3    // methods of the StringBuffer class.
4    import java.awt.Graphics;

```

```
5  import java.applet.Applet;
6
7  public class StringBufferInsert extends Applet {
8      Object o = "hello"; // Assign String to Object reference
9      String s = "good bye";
10     char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
11     boolean b = true;
12     char c = 'Z';
13     int i = 7;
14     long l = 10000000;
15     float f = 2.5f;
16     double d = 33.333;
17     StringBuffer buf;
18
19     public void init()
20     {
21         buf = new StringBuffer();
22     }
23
24     public void start()
25     {
26         buf.insert( 0, o );
27         buf.insert( 0, ' ' );
28         buf.insert( 0, s );
29         buf.insert( 0, ' ' );
30         buf.insert( 0, charArray );
31         buf.insert( 0, ' ' );
32         buf.insert( 0, b );
33         buf.insert( 0, ' ' );
34         buf.insert( 0, c );
35         buf.insert( 0, ' ' );
36         buf.insert( 0, i );
37         buf.insert( 0, ' ' );
38         buf.insert( 0, l );
39         buf.insert( 0, ' ' );
40         buf.insert( 0, f );
41         buf.insert( 0, ' ' );
42         buf.insert( 0, d );
43     }
44
45     public void paint( Graphics g )
46     {
47         g.drawString( "buf = " + buf.toString(), 25, 25 );
48     }
49 }
```

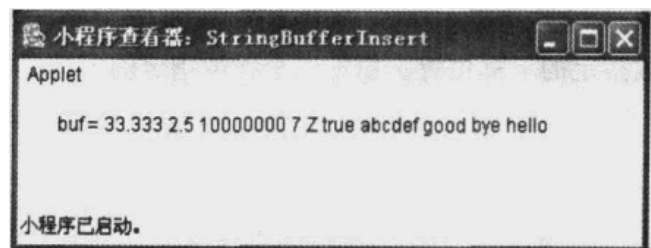


图 8.16 StringBuffer 类的 insert 方法

## 8.19 Character 类的例子

Java提供了许多可将基本数据类型视为对象的类,包括Boolean、Character、Double、Float、Integer和Long。这些类均派生自Object,这就是所谓的类型包装类(type wrapper),它们是Java.lang软件包的一部分。这些类的对象可在程序中需要使用Object或Number的任意位置使用。在本节中,我们将介绍Character类,即用于字符的类型包装类。

大多数Character类的方法是static方法,接收至少一个字符参数,并完成对字符的测试或者对字符的操作。该类也包含以一个char为参数的构造函数来初始化一个Character对象,并且包含几个非static方法。在下面三个例子中提供了Character类的大多数方法,关于Character类(及所有的包装类)的更多信息,请参见Java API文档中的Java.lang软件包。

图8.17中的程序展示了测试字符来判断它们是否是一个特定字符类型的static方法,以及字符大小写转换的static方法。每种方法都用在applet StaticCharMethods类的paint方法中。该程序允许用户输入任意字符,并在该字符上使用上述的方法。

```
1 // Fig. 8.17: StaticCharMethods.java
2 // Demonstrates the static character testing methods
3 // and case conversion methods of class Character
4 // from the java.lang package.
5 import java.awt.*;
6 import java.applet.Applet;
7
8 public class StaticCharMethods extends Applet {
9     char c;
10    Label prompt;
11    TextField input;
12
13    public void init()
14    {
15        c = 'A';
16
17        prompt = new Label( "Enter a character and press Enter" );
18        input = new TextField( "A", 5 );
19        add( prompt );
20        add( input );
21    }
22
23    public void paint( Graphics g )
24    {
25        g.drawString( "is defined: " +
26            Character.isDefined( c ), 25, 40 );
27        g.drawString( "is digit: " +
28            Character.isDigit( c ), 25, 55 );
29        g.drawString( "is Java letter: " +
30            Character.isJavaLetter( c ), 25, 70 );
31        g.drawString( "is Java letter or digit: " +
32            Character.isJavaLetterOrDigit( c ), 25, 85 );
33        g.drawString( "is letter: " +
34            Character.isLetter( c ), 25, 100 );
35        g.drawString( "is letter or digit: " +
36            Character.isLetterOrDigit( c ), 25, 115 );
```

```

37      g.drawString( "is lower case: " +
38                  Character.isLowerCase( c ), 25, 130 );
39      g.drawString( "is upper case: " +
40                  Character.isUpperCase( c ), 25, 145 );
41      g.drawString( "to upper case: " +
42                  Character.toUpperCase( c ), 25, 160 );
43      g.drawString( "to lower case: " +
44                  Character.toLowerCase( c ), 25, 175 );
45  }
46
47  public boolean action( Event e, Object o )
48  {
49      String s = o.toString();
50      c = s.charAt( 0 );
51      repaint();
52      return true;
53  }
54  }

```

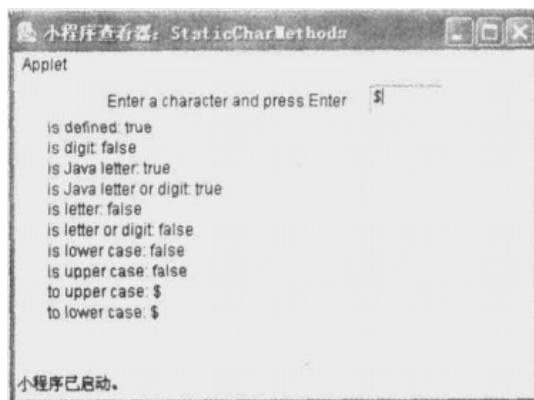
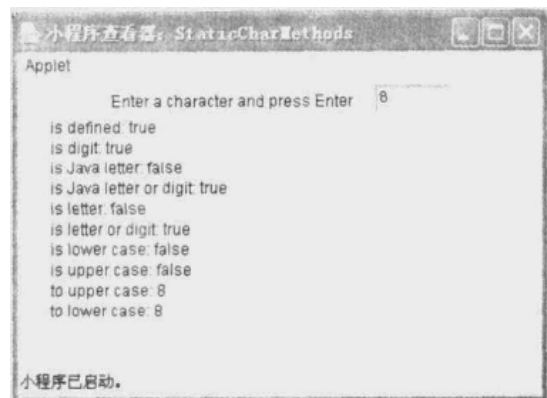
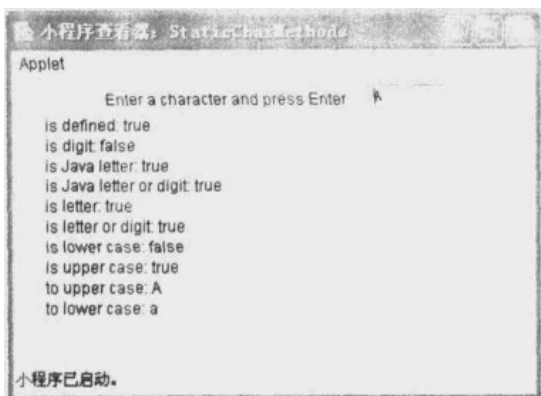


图 8.17 Character 类的 static 字符测试方法和大小写转换方法

下列表达式:

```
Character.isDefined( c )
```

使用 Character.isDefined 方法来判断字符 c 在 Unicode 字符集中是否是已定义的。如果是, 该方法就返回 true; 否则返回 false。

下列表达式:

```
Character.isDigit( c )
```

使用 `Character.isDigit` 方法来判断字符 `c` 是否定义为 Unicode 的数字。如果是就返回 `true`；否则返回 `false`。

下列表达式：

```
Character.isJavaLetter( c )
```

使用 `Character.isJavaLetter` 方法来判断字符 `c` 是否可作为 Java 中标识符的首字符，即字母、下划线 (`_`) 或美元符号 (`$`)。如果是，该方法就返回 `true`；否则返回 `false`。

下列表达式：

```
Character.isLetterOrDigit( c )
```

使用 `Character.isLetterOrDigit` 方法来判断字符 `c` 是否是字母或数字。如果是，则该方法返回 `true`；否则返回 `false`。

下列表达式：

```
Character.isLetter( c )
```

使用 `Character.isLetter` 方法来判断字符 `c` 是否是字母。如果是，则方法返回 `true`；否则返回 `false`。

下列表达式：

```
Character.isLowerCase( c )
```

使用 `Character.isLowerCase` 方法来判断字符 `c` 是否一个小写字母。如果是，则该方法返回 `true`；否则返回 `false`。

下列表达式：

```
Character.isUpperCase( c )
```

使用 `Character.isUpperCase` 方法来判断字符 `c` 是否一个大写字母。如果是，则方法返回 `true`；否则返回 `false`。

下列表达式：

```
Character.toUpperCase( c )
```

使用 `Character.toUpperCase` 方法将字符 `c` 转换成它的大写字符。如果该字符有大写形式，则方法返回转换过的字符；否则返回其原始参数。

下列表达式：

```
Character.toLowerCase( c )
```

使用 `Character.toLowerCase` 方法将字符 `c` 转换成它的小写字符。该方法如果字符有小写形式，则方法返回转换过的字符；否则返回其原始参数。

图 8.18 展示了 `static Character digit` 和 `forDigit` 方法在不同数值系统中的字符和数字之间完成转换的过程。一般的数值系统包括十进制（基数为 10）、八进制（基数为 8）、十六进制（基数为 16）和二进制（基数为 2）。一个数的基数也称为数基（`radix`），如果要获得关于数值系统间进行转换的更多信息，请参见附录 C。

```
1 // Fig. 8.18: StaticCharMethods2.java
2 // Demonstrates the static character conversion methods
3 // of class Character from the java.lang package.
4 import java.awt.*;
5 import java.applet.Applet;
6
7 public class StaticCharMethods2 extends Applet {
8     char c;
9     int digit, radix;
10    boolean charToDigit;
11    Label prompt1, prompt2;
12    TextField input, radixField;
13    Button toChar, toInt;
14
15    public void init()
16    {
17        c = 'A';
18        radix = 16;
19        charToDigit = true;
20
21        prompt1 = new Label( "Enter a digit or character " );
22        input = new TextField( "A", 5 );
23        prompt2 = new Label( "Enter a radix " );
24        radixField = new TextField( "16", 5 );
25        toChar = new Button( "Convert digit to character" );
26        toInt = new Button( "Convert character to digit" );
27        add( prompt1 );
28        add( input );
29        add( prompt2 );
30        add( radixField );
31        add( toChar );
32        add( toInt );
33    }
34
35    public void paint( Graphics g )
36    {
37        if ( charToDigit )
38            g.drawString( "Convert character to digit: " +
39                Character.digit( c, radix ), 25, 125 );
40        else
41            g.drawString( "Convert digit to character: " +
42                Character.forDigit( digit, radix ), 25, 125 );
43    }
44
45    public boolean action( Event e, Object o )
46    {
47        if ( e.target == toChar ) {
48            charToDigit = false;
49            digit = Integer.parseInt( input.getText() );
50            radix = Integer.parseInt( radixField.getText() );
51            repaint();
52        }
53        else if ( e.target == toInt ) {
54            charToDigit = true;
55            String s = input.getText();
```



```

56         c = s.charAt( 0 );
57         radix = Integer.parseInt( radixField.getText() );
58         repaint();
59     }
60
61     return true;
62 }
63 }

```

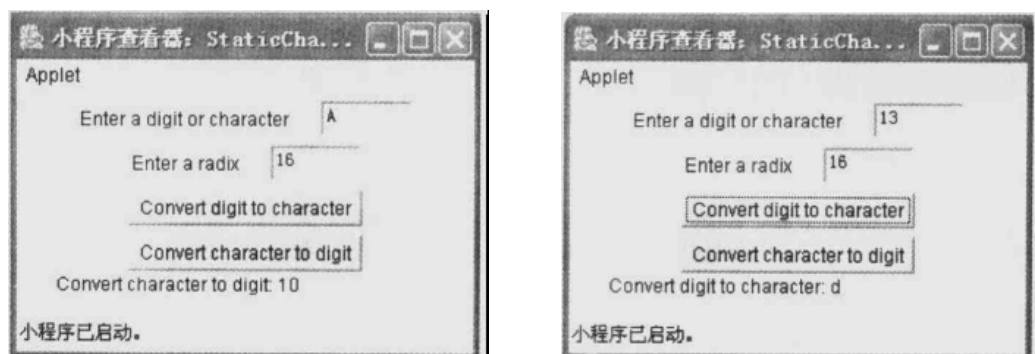


图 8.18 Character 类的 static 转换方法

下列表达式:

```
Character.digit( c, radix)
```

使用 digit 方法将字符 c 转化成由整数 radix 定义的数值系统中的一个整数。例如，字符“A”在基数为 16 的系统中含有整数值 10。

下列表达式:

```
Character.forDigit( digit , radix )
```

使用 forDigit 方法将整数 digit 转换成由整数 radix 说明的数值系统中的一个字符。例如，基数为 16 的整数 13 含有值 'd'。注意，小写字母和大写字母在数值系统中是相同的。

图 8.19 的程序展示了 Character 类的非 static 方法——构造函数 charValue、toString、hashCode 和 equals。

```

1  // Fig. 8.19: OtherCharMethods.java
2  // Demonstrate the non-static methods of class
3  // Character from the java.lang package.
4  import java.awt.Graphics;
5  import java.applet.Applet;
6
7  public class OtherCharMethods extends Applet {
8      Character c1, c2;
9
10     public void init()
11     {
12         c1 = new Character( 'A' );
13         c2 = new Character( 'a' );
14     }
15
16     public void paint( Graphics g )

```

```
17     {
18         g.drawString( "c1 = " + c1.charValue(), 25, 25 );
19         g.drawString( "c2 = " + c2.toString(), 25, 40 );
20         g.drawString( "hash code for c1 = " +
21                     c1.hashCode(), 25, 70 );
22         g.drawString( "hash code for c2 = " +
23                     c2.hashCode(), 25, 85 );
24
25         if ( c1.equals( c2 ) )
26             g.drawString( "c1 and c2 are equal", 25, 115 );
27         else
28             g.drawString( "c1 and c2 are not equal", 25, 115 );
29     }
30 }
```

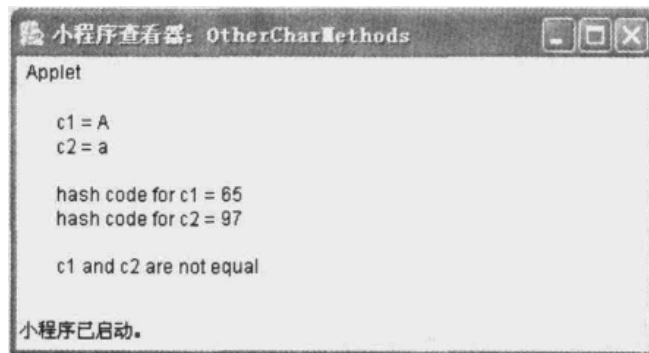


图 8.19 Character 类的非 static 方法

下列语句:

```
c1=new Character( 'A' );
c1=new Character( 'a' );
```

在 OtherCharMethods 类的 init 方法中实例化了两个 Character 对象,并向构造函数传递了类型 char 的常量来初始化该 Character 对象。

下列表达式:

```
c1.charValue( )
```

使用 Character 方法 charValue ( 返回一个 char ) 来获取 Character 对象 c1 的基本数据类型 char 的值。

下列表达式:

```
c2.toString( )
```

使用 Character 方法 toString 返回该 Character 对象 c2 的一个 String 表示。

下列表达式:

```
c1.hashCode( )
c2.hashCode( )
```

分别对 Character 对象 c1 和 c2 完成 hashCode 计算。记住,散列代码值用于将对象存放在易于快速查找的散列表中 ( 详细内容请参见第 18 章 )。

下列条件:

```
c1.equals( c2 )
```

在第25行的if结构中使用equals方法来判断对象c1是否与对象c2的内容相同（即每个对象中的字符相同）。

## 8.20 StringTokenizer 类

在我们阅读书籍的时候，可以将句子分为单个的词汇或语法标记（token），每个标记都向读者传达了某种具体的含义。编译器也实行标记化，它们将语句分成像关键字、标识符、运算符和其他编程语言元素之类的单个部分。在本节中，我们将介绍Java的StringTokenizer类（在Java.util软件包中），它将一个字符串分成组件标记。

语法标记由分隔符彼此分开，典型的分隔符是空白字符，如空格、制表符、换行符和回车符，当然其他字符也可以作为分隔符来分隔语法符号。

图8.20的程序展示了StringTokenizer类。applet Tokentest类显示了一个TextField，在这里用户可以键入要标记化语句。这个程序的输出将传递给另一个TextArea。文本区域类似于文本字段——它们都用于显示或输入文本。该文本区域由下列语句进行实例化：

```
output = new TextArea( 10, 30 )
```

它表明文本区域有10行、30列。我们将在第11章详细讨论文本区域。

```
1 // Fig. 8.20: TokenTest.java
2 // Testing the StringTokenizer class of the java.util package
3 import java.util.*;
4 import java.awt.*;
5 import java.applet.Applet;
6
7 public class TokenTest extends Applet {
8     // GUI components
9     Label prompt;
10    TextField input;
11    TextArea output;
12
13    public void init()
14    {
15        prompt = new Label( "Enter a sentence and press Enter" );
16        input = new TextField( 50 );
17        output = new TextArea( 10, 30 );
18        output.setEditable( false );
19        add( prompt );
20        add( input );
21        add( output );
22    }
23
24    public boolean action( Event e, Object o )
25    {
26        String stringToTokenize = o.toString();
27        StringTokenizer tokens =
28            new StringTokenizer( stringToTokenize );
29
30        output.setText( "" );
```

```

31
32         output.appendText( "Number of elements: " +
33             tokens.countTokens() + " \n The tokens are: \n" );
34
35         while ( tokens.hasMoreTokens() )
36             output.appendText( tokens.nextToken() + " \n" );
37
38         return true;
39     }
40 }

```

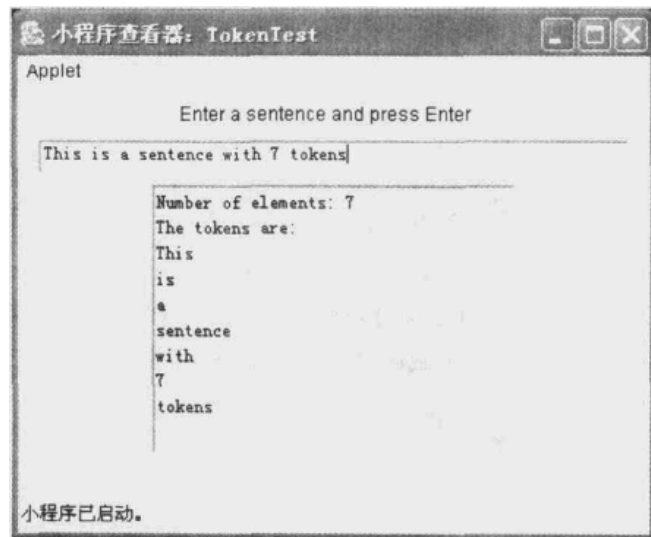


图 8.20 使用 StringTokenizer 对象标记化字符串

当在文本区域按下回车键时，就将激活 action 方法。下列语句：

```
String stringToTokenizer = o.toString();
```

创建了 String 引用 stringToTokenizer，并将 o.toString()——即文本字段中的文本赋给它。接着，创建一个 StringTokenizer 对象，如下所示：

```
StringTokenizer tokens = new StringTokenizer (stringToTokenizer);
```

该语句使用以一个 String 为参数的 StringTokenizer 构造函数来创建 StringTokenizer 对象，它将使用默认的分隔符 “\n\t\r”，这些分隔符是由用于标记化的空白字符、换行符、制表符和回车符组成的。另外，StringTokenizer 还有两个构造函数。在以两个 String 为参数的版本中，第二个 String 为分隔符 String。在三个参数的版本中，第二个 String 为分隔符 String，第三个参数（一个 boolean）判断分隔符是否也作为符号返回（只有当参数为 true 时），这可用于确定分隔符是什么。下列语句：

```
output.setText ( "" );
```

将文本区域中的文本设置为空的 String (""），从而清除了文本区域。

下列语句：

```
output.appendText( " Number of elements:" +
    tokens.countTokens() + " \n The tokens are:\n" );
```

使用 TextArea 的方法 appendText 将作为参数的要连接的 String 附加到文本区域的文本后面。在上面



```

19         "King" ;
20     String suits [] = { "Hearts", "Diamonds",
21         "Clubs", "Spades" };
22
23     deck = new Card[ 52 ];
24     currentCard = -1;
25
26     for ( int i = 0; i < deck.length; i++ )
27         deck[ i ] = new Card( faces[ i % 13 ],
28             suits[ i / 13 ] );
29
30     dealButton = new Button( "Deal card" );
31     shuffleButton = new Button( "Shuffle cards" );
32     displayCard = new TextField( 20 );
33     displayCard.setEditable( false );
34     add( dealButton );
35     add( shuffleButton );
36     add( displayCard );
37 }
38
39 public boolean action( Event event, Object obj )
40 {
41     if ( event.target == dealButton ) {
42         Card dealt = dealCard();
43
44         if ( dealt != null ) {
45             displayCard.setText( dealt.toString() );
46             showStatus( "Card #: " + currentCard );
47         }
48         else {
49             displayCard.setText( "NO MORE CARDS TO DEAL" );
50             showStatus( "Shuffle cards to continue" );
51         }
52     }
53     else if ( event.target == shuffleButton ) {
54         displayCard.setText( "SHUFFLING ..." );
55         showStatus( "" );
56         shuffle();
57         displayCard.setText( "DECK IS SHUFFLED" );
58     }
59
60     return true;
61 }
62
63 public void shuffle()
64 {
65     currentCard = -1;
66
67     for ( int i = 0; i < deck.length; i++ ) {
68         int j = (int) ( Math.random() * 52 );
69         Card temp = deck[ i ];
70         deck[ i ] = deck[ j ];
71         deck[ j ] = temp;
72     }
73
74     dealButton.enable();

```

```

75     }
76
77     public Card dealCard()
78     {
79         if ( ++currentCard < deck.length )
80             return deck [ currentCard ];
81         else {
82             dealButton.disable();
83             return null;
84         }
85     }
86 }
87
88 class Card {
89     private String face;
90     private String suit;
91
92     public Card( String f, String s )
93     {
94         face = f;
95         suit = s;
96     }
97
98     public String toString() { return face + " of " + suit; }
99 }

```



图 8.21 模拟发牌和洗牌

DeckOfCards 类包含一个有 52 张 Card 的数组 deck，一个整数 currentCard 用来代表在该数组中最新发出的牌（如果无牌发出则为 -1），以及用于操作该副牌的 GUI 组件。applet 的 init 方法实例化该 deck 数组（第 26 行），并使用了如下的 for 结构：

```

for ( int i = 0 ; i < deck.length ; i++ )
    deck [ i ] = new Card( faces [ i % 13 ], suit [ i / 13 ] );

```

从而通过 Card 来填充 deck 数组。注意, 每个 Card 都将实例化, 并由两个 String 进行实例化——一个来自 faces 数组 (String “Ace” 到 “King”), 一个来自 suits 数组 (“Hearts”、“Diamonds”、“Clubs” 和 “Spades”)。算式 “ $i \% 13$ ” 总是产生一个 0 到 12 的值 (faces 数组的 13 个下标), 算式 “ $i/13$ ” 总是产生一个 0 到 3 的值 (suits 数组的 4 个下标) 当初始化 deck 数组时, 对于每个花色都包含从 A 到 K 的牌点。

当用户单击 “Deal card” 按钮时, action 激活 dealcard 方法 (第 42 行), 从数组中获取下一张牌。如果 deck 非空, 就返回一个 Card 对象, 否则返回 null。如果引用不是 null, 则执行下列语句:

```
displayCard.setText ( dealt.toString ( ) );  
showStatus ( " Card # : " +currentCard );
```

在文本字段 displayCard 中显示 Card, 同时在 applet 的状态栏上显示牌点。

如果 dealCard 返回的引用为 null, 则在文本字段中显示 String “NO MORE CARDS TO DEAL”, 同时在 applet 的状态栏上显示 String “Shuffle cards to continue”

当用户单击 “Shuffle cards” 按钮时, action 方法激活 shuffle() 方法 (第 63 行) 进行洗牌。该方法遍历所有 52 张牌 (数组下标从 0 到 51), 对于每一张牌而言, 可以任意选取 0 到 51 的号码。接着, 当前的 Card 对象和任意选择的 Card 对象在数组中进行对换, 对于整个数组的单遍扫描只需 52 次对换, 这样就洗完了 Card 对象数组。当洗牌过程结束后, 在文本字段中显示 String “DECK IS SHUFFLED”。

## 小结

- 字符常量的值是它在 Unicode 字符集中的整数值。字符串可以包含字母、数字和特殊字符, 例如 +、-、\*、/ 和 \$。Java 中的一个字符串是 String 类的一个对象, 字符串直接量或字符串常量通常称为匿名 String 对象, 在一个程序中用双引号括起来。
- String 类提供了 7 个构造函数, 它们用来将 String 初始化为空、复制当前的 String、复制来自另一个字符数组的字符 (或字符的一部分)、复制字节 (或字节数组中字节的一部分) 或者复制 StringBuffer 中的字符。
- String 类的 length 方法用于返回 String 中的字符个数。
- String 类的 charAt 方法用于在一个特定位置选取字符, getBytes 方法用于把 String 的字符复制到一个 byte 数组。
- String 类的 equals 方法 (由 String 类从其超类 Object 继承) 用于测试任意两个对象的相等性 (即两个对象的内容是相同的), 如果对象相等则方法返回 true, 否则返回 false。equals 方法使用词典比较法。
- 运算符 “==” 在用于比较引用和用于比较基本数据类型时具有不同的功能。当使用 “==” 比较基本数据类型值时, 如果值相同则结果为 true。当使用 “==” 比较引用时, 如果引用内存中的相同对象则结果为 true。
- Java 将含有相同内容的匿名 String 视为有许多引用的同一个匿名 String 对象。
- String 类的 equalsIgnoreCase 方法在执行一次比较时忽略每个 String 中字母的大小写。
- 如果所比较的 Strings 相等, 则 String 类的 compareTo 方法返回 0; 如果激活 compareTo 方法的 String 小于作为参数的 String, 则返回一个负数; 如果激活 compareTo 方法的 String 大于作为参数的 String, 则返回一个正数。compareTo 方法使用词典比较法。



- String类的regionMatches方法用于比较两个String的部分相等性。
- String类的startsWith方法用于判断调用它的String对象是否以参数说明的字符开始。
- String类的endsWith方法用于判断调用它的String对象是否以参数说明的字符结尾。
- String类的hashCode方法用于执行散列代码计算,使一个String对象能存入散列表中。Object的所有子类将继承这一方法,并由相应的String覆盖。
- String类的indexOf方法用于定位String中某个字符首次出现的位置,该方法有一个版本可在一个String中查找子字符串。lastIndexOf方法用于定位String中某个字符最后一次出现的位置,这个方法也有一个版本可用于在String中查找子字符串。
- String类的substring方法可以通过复制现有String对象的一部分而创建一个新的String。
- String类的concat方法可用于连接两个String对象并返回一个新的String对象,其中包含来自两个原始String的字符。
- String类的replace方法可用于返回一个新的String对象,它将String中出现的所有第一个字符参数都替换成第二个字符参数,同时不改动原始String。
- String类的toUpperCase方法用于返回一个新的String对象,将原始String中的小写字母都换成大写字母。toLowerCase方法用于返回一个新的String对象,将原始String中的大写字母都换成小写字母,同时不改动原始String。
- String类的trim方法用于返回一个新String对象,从String的开头或结尾删去所有的空白字符(如空格、换行符和制表符),同时不改动原始String。
- String类的toArray方法创建了一个包含String中字符副本的新字符数组,并返回这个新数组。
- String类的valueOf方法将参数转换成字符串并作为String对象返回。
- 第一次在String对象上调用String类的intern方法时,该方法返回这个String对象在内存中的引用。随后在与原始String有相同内容的不同String对象上调用intern时,都将导致对原始String对象的多重引用。
- StringBuffer类提供了3个构造函数,使得能将StringBuffer初始化成3种情况:无字符并具有初始的16个字符容量;无字符和由整数参数指定的初始容量;或者含有String参数的一份字符副本,以及初始的String参数中的字符数加上16的容量。
- StringBuffer类的length方法用于返回当前存储在StringBuffer中的字符数,capacity方法用于返回StringBuffer中可以存储的字符数,并且不会分配更多的内存。
- ensureCapacity方法用于保证StringBuffer具有最小容量, setLength方法用于增加或减少StringBuffer的长度。
- StringBuffer类的charAt方法用于返回指定位置上的字符, setCharAt方法用于在指定位置上设置字符, getChars方法用于返回一个包含StringBuffer中字符副本的字符数组。
- StringBuffer类提供了重载的append方法,允许将各种数据类型值添加到一个StringBuffer的末尾,并对每种基本数据类型、字符数组、String和Object都提供了相应版本。
- Java编译器将StringBuffer的append方法用于实现连接String的“+”和“+=”运算符。
- StringBuffer类提供了重载的insert方法,允许将各种数据类型值插入到StringBuffer的任意位置上,针对每种基本数据类型、字符数组、String和Object都提供了相应的版本。
- Character类提供了一个以字符为参数的构造函数。
- Character类的isDefined方法用于判断一个字符是否在Unicode字符集中已经定义。如果有,则返回true;否则返回false。
- Character类的isDigit方法用于判断一个字符是否是一个已定义的Unicode数字。如果是,则

该方法返回 true；否则返回 false。

- Character 类的 isJavaLetter 方法用于判断一个字符是否可作为 Java 中标识符的首字符，即字母、下划线 ( \_ ) 或美元符号 ( \$ )。如果是，则该方法返回 true；否则返回 false。
- Character 类的 isJavaLetterOrDigit 方法用于判断一个字符是否可作为 Java 中的标识符，即数字、字母、下划线 ( \_ ) 或美元符号 ( \$ )。如果是，则该方法返回 true；否则返回 false。
- Character 类的 isLetter 方法用于判断一个字符是否是字母。如果是，则该方法返回 true；否则返回 false。
- Character 类的 isLetterOrDigit 方法用于判断一个字符是否是字母或数字。如果是，则该方法返回 true；否则返回 false。
- Character 类的 isLowerCase 方法用于判断一个字符是否是字母。如果是，则该方法返回 true；否则返回 false。
- Character 类的 isUpperCase 方法用于判断一个字符是否是大写字母。如果是，则该方法返回 true；否则返回 false。
- Character 类的 toUpperCase 方法用于将一个字符转换成其相应的大写字母，toLowerCase 方法则用于将一个字符转换成其相应的小写字母。
- Character 类的 digit 方法用于将它的字符参数转换成由它的整数参数 radix (即基数) 所指定的数值系统中的一个整数，forDigit 方法用于将它的整数参数 digit 转换成由其整数参数 radix 指定的数值系统中的一个字符。
- charValue 方法用于返回存储在一个 Character 对象中的字符，toString 方法用于返回一个 Character 的 String 表示。
- hashCode 方法用于在 Character 上执行散列代码计算。
- StringTokenizer 的默认构造函数为它的 String 参数创建一个 StringTokenizer，它将使用由空白字符、换行符、制表符以及回车符组成的默认分隔符字符串 “\n \t \r” 来进行标记化。
- 在以两个 String 为参数的 StringTokenizer 构造函数中，第二个 String 是分隔符 String。在有三个参数的 StringTokenizer 构造函数中，第二个 String 是分隔符 String，而第三个参数 (一个 boolean) 决定分隔符是否也作为符号返回。
- StringTokenizer 方法 countTokens 用于返回要标记化的 String 中的标记数目。
- StringTokenizer 方法 hasMoreTokens() 用于判断正在标记化的 String 中是否还有标记。
- StringTokenizer 方法 nextToken 用于返回带有下一个标记的 String。

## 术语

|                                       |              |                                     |                  |
|---------------------------------------|--------------|-------------------------------------|------------------|
| append method of class StringBuffer   | StringBuffer | character code                      | 字符编码             |
| 类的 append 方法                          |              | character constant                  | 字符常量             |
| appending strings to other strings    | 向其他字符串       | character set                       | 字符集              |
| 附加字符串                                 |              | charAt method of class String       | String 类的 charAt |
| array of strings                      | 字符串的数组       | 方法                                  |                  |
| capacity method of class StringBuffer | StringBuffer | charAt method of class StringBuffer | StringBuffer 类   |
| 类的 capacity 方法                        |              | 的 charAt 方法                         |                  |
| Character class                       | Character 类  | charValue method of class Character | Character 类      |

- 的 charValue 方法
- compareTo method of class String String类的  
compareTo 方法
- comparing strings 比较字符串
- concat method of class String String类的 concat  
方法
- concatenation 连接
- copying strings 复制字符串
- countTokens method of StingTokenizer  
StringTokenizer类的 countTokens 方法
- delimiter 分隔符
- digit method of class Character Character类的digit  
方法
- endsWith method of class String String类的  
endsWith 方法
- equals method of class String String类的 equals  
方法
- equalsIgnoreCase method of String String类的  
equalsIgnoreCase 方法
- forDigit method of class Character Character类的  
forDigit 方法
- getBytes method of class String String类的getBytes  
方法
- getChars method of class String String类的getChars  
方法
- getChars method of class StringBuffer StringBuffer  
类的 getChars 方法
- hashCode method of class Character Character类  
的 hashCode 方法
- hashCode method of class String String类的  
hashCode 方法
- hash table 散列表
- hasMoreTokens method hasMoreTokens 方法
- hexadecimal digits 十六进制数字
- high-order byte 高位字节
- indexOf method of class String String类的indexOf  
方法
- insert method of class StringBuffer StringBuffer类  
的 insert 方法
- intern method of class String String类的 intern  
方法
- isDefined method of class Character Character类  
的 isDefined 方法
- isDigit method of class Character Character类的  
isDigit 方法
- isJavaLetter method of class Character Character  
类的 isJavaLetter 方法
- isJavaLetterOrDigit method isJavaLetterOrDigit方法
- isLetter method of class Character Character类的  
isLetter 方法
- isLetterOrDigit method of Character Character类  
的 isLetterOrDigit 方法
- isLowerCase method of class Character Character  
类的 isLowerCase 方法
- isUpperCase method of class Character Character  
类的 isUpperCase 方法
- lastIndexOf method of class String String类的  
lastIndexOf 方法
- length method of class String String类的length方  
法
- length method of class StringBuffer StringBuffer类  
的 length 方法
- length of a string 字符串的长度
- literal 直接量
- low-order byte 低位字节
- nextTokens method of class StringTokenizer  
StringTokenizer类的 nextTokens 方法
- numeric code representation of a character 字符的  
数字编码表示
- printing character 打印字符
- regionMatches method of calss String String类的  
regionMatches 方法
- replace method of class String String类的replace  
方法
- search string 查找字符串
- setCharAt method of class StringBuffer StringBuffer  
类的 setCharAt 方法
- startsWith method of class String String类的  
startsWith 方法
- string 字符串
- String class String类
- string concatenation 字符串连接

|                                                                   |                                                                   |
|-------------------------------------------------------------------|-------------------------------------------------------------------|
| string constant 字符串常量                                             | toString method of class Character Character 类的 toString 方法       |
| string literal 字符串直接量                                             | toString method of class String String 类的 toString 方法             |
| string processing 字符串处理                                           | toString method of class StringBuffer StringBuffer 类的 toString 方法 |
| StringBuffer class StringBuffer 类                                 | toUpperCase method of class Character Character 类的 toUpperCase 方法 |
| StringIndexOutOfBoundsException                                   | toUpperCase method of class String String 类的 toUpperCase 方法       |
| StringTokenizer class StringTokenizer 类                           | trim method of class String String 类的 trim 方法                     |
| substring method of String class String 类的 substring 方法           | Unicode 国际标准字符集                                                   |
| toCharArray method of class String String 类的 toCharArray 方法       | valueOf method of class String String 类的 valueOf 方法               |
| token 语法标记                                                        | white space characters 空白字符                                       |
| tokenizing strings 标记化字符串                                         | word processing 字处理                                               |
| toLowerCase method of class Character Character 类的 toLowerCase 方法 |                                                                   |
| toLowerCase method of class String String 类的 toLowerCase 方法       |                                                                   |

## 自测练习

- 8.1 判断下列语句是否正确。如果不正确，请解释原因。
- 当 String 对象用 “==” 比较时，如果 String 包含相同的值则结果为 true。
  - 一个 String 在其创建后可以修改。
- 8.2 对于下列描述，编写一条语句完成要求的任务。
- 比较 s1 中的字符串和 s2 中的字符串的内容相等性。
  - 用 “+=” 向字符串 s1 附加字符串 s2。
  - 判断 s1 中字符串的长度。

## 自测练习答案

- 8.1 a) 不正确。用 “==” 运算符比较 String 对象实际上是判断它们在内存中是否为同一对象。  
b) 不正确。String 对象是常量，在它们创建后不能修改。
- 8.2 a) `s1.equals(s2);`  
b) `s1 += s2;`  
c) `s1.length();`

## 练习

[注意：练习 8.3 到练习 8.6 具有相当大的挑战性。解决了这些问题后，就可以轻易地实现大多数流行的纸牌游戏。]

- 8.3 修改图 8.21 中的程序，以便发牌方法能一手发 5 张牌。然后写出下列附加的方法：

- a) 判断该手牌是否含有一对牌。
  - b) 判断该手牌是否含有两对牌。
  - c) 判断该手牌是否含有3张同花色的牌(例如3张J)。
  - d) 判断该手牌是否含有4张同花色的牌(例如4张A)。
  - e) 判断该手牌是否含有同花色的5张牌(即5张同一花色的牌)。
  - f) 判断该手牌是否含有一条龙(即5张连续点数的牌)。
- 8.4 使用练习8.3中开发的方法,编写一个程序发出两手5张牌,评价每一手牌,判断哪一手更好。
- 8.5 修改练习8.4中开发的程序以使它能模拟发牌者。发牌者的一手5张牌均面朝下发出,因此玩家看不见这手牌。然后程序评价发牌者的这手牌,发牌者应当挑出1张、2张或3张牌来替换原先该手牌中不需要的相同数目的牌,然后程序再评价发牌者的这手牌。注意:这是个难题。
- 8.6 修改练习8.5中开发的程序,以便它能自动处理发牌者的一手牌,但是玩家可以决定手中的哪些牌要换。程序随后评价这两手牌并判定谁赢。现在,利用这个新程序同计算机玩20局。谁赢得多些?根据这些游戏的结果,对程序进行适当的修改,以使纸牌游戏程序更加完善(这也是一个难题)。再玩20局,修改后的程序性能更好了吗?
- 8.7 编写一个applet,利用String类的compareTo方法来比较两个由用户输入的字符串,程序应当指出第一个字符串是否小于、等于或大于第二个字符串。
- 8.8 编写一个applet,利用regionMatches方法来比较由用户输入的两个字符串。程序应当输入要比较的字符个数和比较的开始下标。程序应当指出第一个字符串是否小于、等于或大于第二个字符串。在执行比较时忽略字符的大小写。
- 8.9 编写一个applet,利用随机数产生器来创建语句。使用称为article、noun、verb和preposition的4个数组,按照下列顺序从每个数组中选取随机的单词来创建一条语句: article、noun、verb、preposition、article和noun。选出每个单词后,将其连接到语句中的前一个单词之后,单词之间应当由空格分开。输出最后的语句时,该语句应当以大写字母开头并以句点结尾。程序应当生成20个语句并输出到一个文本区域中。

数组的内容如下:

article 数组应包含冠词 the、a、one、some 和 any;  
noun 数组应当包含名词 boy、girl、dog、town 和 car;  
verb 数组应当包含动词 drove、jumped、ran、walked 和 skipped;  
preposition 数组应当包含介词 to、from、over、under 和 on。

编写完上面的程序后,修改该程序以生成包含这几个句子的小故事。

- 8.10 (5行打油诗) 5行打油诗是一首幽默的5五行诗,第一行和第二行与第五行押韵,第三行同第四行押韵。使用类似练习8.9中的开发方法,编写一个Java程序,随机生成5行打油诗。
- 8.11 编写一个applet,将英语单词编码成pig Latin。pig Latin是一种常用于娱乐编码的语言形式,有许多方法用于组成pig Latin,简单来说,可以使用下列算法:

为了从英语词组生成pig Latin,可以使用StringTokenizer类的对象来将词组分成单词。为了将每个英语单词翻译成一个pig Latin,可以将英语单词的第一个字母放在单词末尾并加上字母“ay”。这样单词“jump”变成了“umpjay”,单词“the”变成了“hetay”,单

词“computer”变成了“omputercay”，单词之间的空格仍为空格。这里做如下假定：英语词组的单词由空格分开，没有标点符号，所有的单词有两个或更多的字母。应当使用 `printLatinWord` 方法显示每个单词，每个由 `nextToken` 返回的语法标记都传递给 `printLatinWord` 方法来打印 pig Latin。该程序应读入用户输入的语句，并在文本区域中显示所有已转换的语句。

- 8.12 编写一个 applet，利用格式(555)555-5555 将电话号码作为一个字符串输入。程序应当使用 `StringTokenizer` 类的一个对象将区号作为一个语法标记，电话号码的前3位作为一个标记，电话号码的后4位作为另一个标记提取出来，电话号码的7位应当连接成一个字符串。程序应当将区号字符串转换成 `int` 类型，并将电话号码字符串转换成 `long` 类型，然后打印区号和电话号码。记住，需要在标记化过程中改变分隔符。
- 8.13 编写一个 applet，输入一行文本，使用 `StringTokenizer` 类的一个对象将其标记化，并以逆序输出标记。
- 8.14 使用已讨论过的字符串比较方法和第5章中开发的排序数组技术，编写一个程序，按词典顺序对一组字符串进行排序。允许用户在一个文本区域中输入 `String`，并在一个文本区域中显示结果。
- 8.15 编写一个 applet，输入文本，并按大写字母和小写字母输出该文本。
- 8.16 编写一个 applet，输入几行文本和一个查找字符，使用 `String` 类的 `indexOf` 方法来判断文本中该字符出现的次数。
- 8.17 编写一个基于练习 8.16 中程序的 applet，输入几行文本并使用 `String` 类的 `indexOf` 方法来判断文本中每个字母出现的总次数，大写和小写字母应当一同计算。在数组中保存每个字母的统计结果，在确定全部结果后以表格形式打印出值。
- 8.18 编写一个 applet，读入一系列字符串并只打印那些以字母“b”开头的字符串，将结果输出到文本区域中。
- 8.19 编写一个 applet，读一系列字符串并只打印那些以字母“ED”结尾的字符串，将结果输出到文本区域中。
- 8.20 编写一个 applet，输入一个字符的整数编码并显示相应的字符。修改此程序以便它生成0到255之间的所有可能的三位编码，并试图打印相应的字符，在一个文本区域中显示结果。
- 8.21 编写读者自己的查找字符串的 `String` 方法。

## 特殊小节：高级字符串操作练习

前面的练习是针对文本的，目的是要测试读者对基本字符串操作概念的理解。本节包括一组中级和高级字符串操作练习，读者会发现这些问题虽具挑战性但很有意思，问题的难度相差很大。其中一些需要一两个小时来编写程序并实现。另一些对于实验室练习则是很有意义的，需要二周到三周的时间来研究和实现，还有一些是挑战性的学期项目。

- 8.22 (文本分析) 使用计算机的字符串操作功能，可以对大作家的作品进行一些相当有趣的分析。现在，有越来越多的人在关注是否曾有过莎士比亚其人。一些学者相信有确凿的证据表明 *Christopker MarLowe* 或其他作家撰写了莎士比亚作品的主要部分。研究人员已经使用计算机来寻找两位作家作品的相似性。这个练习考察了计算机的3个文本分析方法。

a) 编写一个 applet，从键盘读取几行文本并打印一个表格，表明文本中按字母表顺序的字母出现次数。例如，下列词组：

To be , or not to be : that is the question.

包含一个“a”，两个“b”，没有“c”，等等

- b) 编写一个 applet，读取几行文本并打印一个表格，指明出现一个字母的单词、两个字母的单词、三个字母的单词等的数目。例如，对于下列词组：

Whether 'tis nobler in the mind to suffer

其结果如下：

| 单词长度 | 出现次数        |
|------|-------------|
| 1    | 0           |
| 2    | 2           |
| 3    | 2           |
| 4    | 2 (包括 'tis) |
| 5    | 0           |
| 6    | 2           |
| 7    | 1           |

- c) 编写一个 applet，读取几行文本并打印一个表格，指明文本中每个不同单词的出现次数。其中第一版本的程序应当将单词按照在文本中出现的顺序而安排在表格中。例如，下面两行：

To be , or not to be :that is the question  
Whether 'tis nobler in the mind to suffer

包含单词“to”三次，单词“be”二次，单词“or”二次，等等。一个更有趣的打印输出是尝试先把这些单词按词典顺序排序。

- 8.23 (使用不同的格式打印日期) 日期可以使用几种常见的格式打印。两种较为常见的格式是：

07/21/55 和 July21 ,1955

编写一个 applet，读取第一种格式的日期并用第二种格式打印出该日期。

- 8.24 (支票保护) 计算机常常用于支票打印系统，如工资支付和付账系统。有时可能会产生错误，例如将每周工资打印成超过 100 万美元。因为人为错误或机器失效，计算机化的支票打印系统打印出不正确的数额。为此，系统设计人员在系统中加入控制机制，以防止发出此类错误支票。另一个严重的问题，则是由欺诈性兑现支票的人引起的支票数额的国际更改。为防止更改美元数额，大多数计算机化的支票检查系统使用了称为支票保护 (check protection) 的技术。用于计算机打印的支票包含固定数目的空格，计算机可以在其中打印一个数额。假定一张支付支票有 8 个空格，计算机在其中打印一周的工资。如果数额太大，则将填满 8 个空格。例如：

1,230.60 (支票数额)  
-----  
12345678 (位置号)

另一方面，如果总额小于 \$1 000，那么一般要留下几个空格。例如：

99.87  
-----  
12345678

包含 3 个空格。如果一张支票打印留有空格,就使一些人容易更改支票数额。为了防止更改支票,许多支票打印系统插入前导星号来保护数额,如下所示:

```
*** 99.87
-----
12345678
```

编写一个 applet, 输入一个要在支票上打印的美元数额, 然后使用带有前导星号的支票保护格式来打印这个数额。假定可以在 9 个空格中打印该数额。

- 8.25 (打印支票数额的等额大写单词) 继续讨论前一个练习中的问题, 我们将再次强调设计支票检查系统的重要性, 以防止更改支票数额。一种通常的保护方法是既用数字写出支票数额, 同时又用单词“拼出”该数额。即使某些人能更改支票的数额, 但极难改变大写的数额。许多计算机化的支票打印系统并不用大写形式打印支票总额, 省略的主要原因也许是大多数用于计算机应用的高级语言未包含足够的字符串操作特性, 另一个原因涉及到打印大写支票数额的逻辑。编写一个 applet, 输入一个数字支票总额并打印其数额的大写形式。例如, 数额 112.43 应写成:

ONE HUNDRED TWELVE and 43/100

- 8.26 (莫尔斯码) 在所有的编码体系中, 最著名的也许是莫尔斯码 (Morse Code), 它是由 Samuel Morse 在 1832 年发明的并用于电报系统。莫尔斯码将字母表中的每个字母、数字以及一些特殊字符 (如句点、分号、冒号等) 赋值为一系列的点和划线。在基于声音的系统中, 点表现为一个短音而划线表现为一个长音。

单词间的分隔符用一个空格来表示, 或者不用点或划线。莫尔斯码的国际版本如图 8.22 所示。

| 字符 | 编码    | 字符     | 编码     |
|----|-------|--------|--------|
| A  | .-    | T      | -      |
| B  | -...  | U      | ..-    |
| C  | -.-.  | V      | ...-   |
| D  | -..   | W      | -.--   |
| E  | .     | X      | -.-.-  |
| F  | ..-.  | Y      | -.--   |
| G  | --.   | Z      | --..   |
| H  | ....  |        |        |
| I  | ..    | Digits |        |
| J  | .-..  | 1      | .-.... |
| K  | -.-   | 2      | ..-..  |
| L  | .-... | 3      | ...--  |
| M  | --    | 4      | ....-  |
| N  | -.    | 5      | .....  |
| O  | ---   | 6      | -....  |
| P  | .-.-. | 7      | --...  |
| Q  | --.-  | 8      | ---..  |
| R  | ..-   | 9      | -----  |
| S  | ...   | 0      | -----  |

图 8.22 字母表中用国际莫尔斯码表示的字母

编写一个 applet, 读取英语词组并将词组编码为莫尔斯码; 然后再编写另一个程序读取莫尔斯码, 并将词组转换成等价的英语词组。每个编码为莫尔斯码的字母间用 1 个空格分开, 每个编码为莫尔斯码的单词间用 3 个空格隔开。



- 8.27 (公制转换程序) 编写一个 applet, 帮助用户进行公制转换。该程序应当允许以字符串方式说明单位的名称(即十进制的厘米、升、克等, 英制的英寸、夸脱、磅等), 并回答如下的简单问题:

```
" How many inches are in 2 meters? "  
" How many liters are in 10 quarts? "
```

该程序应当识别非法的转换。例如, 下列问题:

```
" How many feet in 5 kilograms ? "
```

是没有意义的, 因为“feet”是长度单位, 而“kilogram”是重量单位。

## 挑战性的字符串操作项目

- 8.28 (项目: 拼写检查器) 许多流行的字处理软件包都包含内置的拼写检查器。在本练习中, 要求读者开发自己的拼写检查器实用程序。我们的提示可以帮助读者开始编写程序, 读者应当随后考虑添加更多的功能。程序中使用一个计算机化的词典作为单词源。

我们为什么会输入许多拼写不正确的单词呢? 在某些情况下, 因为我们不知道正确的拼写, 于是进行一个“最佳猜测”。在其他一些情况下, 可能由于颠倒了两个字母(例如“default”而非“default”)。有时我们偶然双击了一个键(例如“hanndy”而非“handy”), 有时我们键入了应输入字母附近的一个键(例如“biryhday”而非“birthday”)。诸如此类, 还有其他一些情况。利用 Java 设计并实现一个拼写检查器, 你的程序应当维护一个字符串的数组 wordList, 使用户能处理这些字符串。[注意: 第 15 章和第 16 章中我们将介绍文件处理和网络, 掌握了这些功能后, 就能从一个计算机化的词典中获取用于拼写检查的单词。]

你的程序应当要求用户输入一个单词, 然后在 wordList 数组中查找这个单词。如果该单词在数组中, 程序应当打印“Word is spelled correctly.”。

如果单词不在数组中, 程序应当打印“Word is not spelled correctly.”, 接着程序应当试着在 wordList 中定位那些也许是用户想要键入的单词。例如, 可以试着找出相邻字母的所有可能的单个字母颠倒, 从而发现单词“default”是 wordList 中一个单词的直接匹配。当然, 这意味着程序将检查所有其他的单个字母颠倒, 如 edfault、dfeault、deafult、defalut 和 defaultl。如果发现了一个新的单词与 wordList 中的某个匹配时, 则打印一条消息“Did you mean “default”?”。

实现其他的一些测试, 诸如用单个字母替换每个双写字母, 读者可以试着改善拼写检查器的功能。

- 8.29 (项目: 纵横填字谜生成器) 大多数人都喜欢玩填字谜(crossword puzzle)游戏, 但几乎没人去尝试生成一个这样的游戏程序。这里建议把生成一个填字谜游戏作为一个需要真正复杂技术和努力的字符串操作项目。程序员需要解决许多问题, 才能让最简单的填字谜生成器运行起来。例如, 如何在计算机中表示一个填字谜游戏的方格? 程序员需要一个单词源(即计算机化的词典), 并可由程序直接引用。使用什么样的格式存储这些单词才能实现程序要求的复杂操作? 真正有兴趣学习的读者会想为每个字谜生成“线索”部分, 其中每个“交叉”单词和每个“垂直下降”单词的简要提示将打印给猜谜者。单是打印一个空的填字谜本身也不简单。

## 第9章 图 形

### 教学目标

- 理解图形环境和图形对象
- 理解并学习绘制字符串、字符和字节的方法
- 理解并学习处理颜色
- 理解并学习处理字体
- 理解并学习绘制线条、矩形、圆角矩形、三维矩形、椭圆、圆弧和多边形
- 学会复制屏幕的区域
- 理解绘图模式

### 9.1 简介

我们现在开始对Java抽象窗口化工具箱（AWT, Abstract Windowing Toolkit）进行深入考察，它们是构成java.awt软件包的类。本章我们将提供在屏幕上进行绘图的图形功能。图9.1显示了java.awt类层次的一部分，其中包括了本章涵盖的一些类，图中的每个类都从Object类直接继承。Color类包含用于操作颜色的方法和常量，Font类包含用于操作字体的方法和常量。FontMetrics类包含获取字体信息的方法，Polygon类包含创建多边形的方法，Graphics类包含绘制字符串、直线、矩形和其他形状的方法，Toolkit类提供了从一个系统（如可显示的字体集和屏幕分辨率）中获取图形信息的方法。

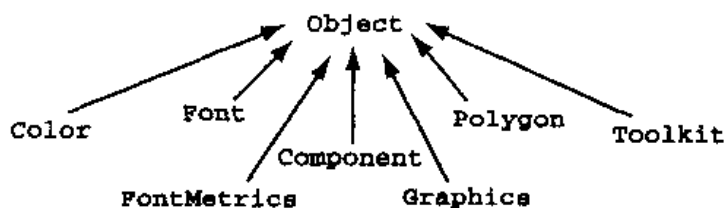


图 9.1 java.awt 层次的一部分

在Java中开始绘图之前，我们必须首先了解Java的坐标系（如图9.2所示），它是一个可以标识屏幕上每个点的系统。默认状态下，屏幕左上角的坐标为(0,0)。一对坐标由一个x坐标（水平坐标）和一个y坐标（垂直坐标）组成。x坐标是从左上角向右移动的水平距离。y坐标是从左上角向下移动的垂直距离。x轴描述了每个水平坐标，而y轴描述了每个垂直坐标。

文本和形状利用一定的坐标来显示在屏幕上，坐标单位用像素来度量。一个像素是一台监视器的最小分辨单位。

#### 可移植性提示 9.1

不同的显示监视器有不同等级的分辨率（即像素变化密度）。

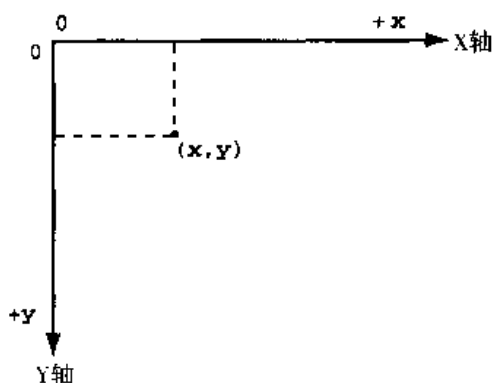


图 9.2 Java 坐标系, 单位用像素来度量

## 9.2 图形环境和图形对象

图形环境 (graphic context) 使 Java 能够在屏幕上绘图。Graphics 对象通过控制如何绘制信息来管理一个图形环境, Graphics 对象包含绘图、字体操作和颜色操作等方法。我们已在前面的内容中看到, 每个 applet 中执行屏幕绘图的程序已经使用了 Graphics 对象 g (applet 的 paint 方法的参数) 来管理 applet 的图形环境。

一个 Graphics 对象必须用于绘图。Graphics 类是一个 abstract 类——即不能实例化 Graphics 对象。这样做的原因是为了 Java 的可移植性。因为在支持 Java 的每个平台上绘图的执行过程都是不同的, 因此不能有一个类来实现所有系统中的绘图功能。例如, 使一台运行 Windows 的计算机绘制一个矩形的图形功能, 就不同于使一台 UNIX 工作站绘制一个矩形的图形功能, 而且它们又都不同于使一台 Macintosh 绘制一个矩形的图形功能。当 Java 在每个平台上实现时, 就将创建一个 Graphics 的派生类, 用来实现所有的绘图功能。Graphics 类将这一实现隐藏起来, 该类提供的接口使我们能够以一种与平台无关的方式编写图形应用程序。

Component 类的 paint 方法以一个 Graphics 对象为参数, 当 Component 类中含有绘图操作时, 系统就将 Graphics 对象传递给 paint 方法。Component 类是 AWT 中许多类的超类, paint 方法的开头为:

```
public void paint (Graphics g )
```

Graphics 对象 g 接收一个系统的派生 Graphics 类对象的引用。前面的方法首部看上去很熟悉——它与我们在 Applet 类中使用的一样。Component 类是一个 Applet 类的间接基类, 后者是由每个 applet 继承的。事实上, Applet 类的许多功能都继承自 Component 类。Component 类中定义的 paint 方法在默认情况下不执行任何操作——必须由程序员重写 (覆盖) 该方法。

paint 方法很少由程序员直接调用, 因为绘制图形是一个事件驱动过程。当开始执行 applet 时, 将自动调用 paint 方法 (在调用 applet 的 init 和 start 方法之后)。如果想再次调用 paint 方法, 则必须要发生一个事件才行, 例如改变 applet 的大小。

如果程序员需要调用 paint 方法, 那就要向 Component 类发出 repaint 方法调用。repaint 方法调用利用 Component 类的 update 方法来清除 Component 以前所绘制的背景, 接着 update 方法将直接调用 paint 方法。程序员经常调用 repaint 方法来强制完成一次 paint 操作, 因此应当重写 repaint 方法。程序中很少直接调用 update 方法, 有时也可以将其重写。重写 update 方法对于“平滑”动画 (即减少跳动) 是很有用的, 我们将在第 14 章中看到这一点。repaint 和 update 方法的开头是:

```
public void repaint ( )  
public void update (Graphics g )
```

两个方法都是 public 类型并具有 void 返回类型。update 方法以 Graphics 对象为参数，它是由 repaint 方法自动提供的。

在本章中我们主要介绍 paint 方法。在下一章中我们将更加专注于图形的事件驱动性质，并详细讨论 repaint 方法和 update 方法。

### 9.3 绘制字符串、字符和字节

本节提供了用于绘制字符串、字符和字节的 Graphics 方法，在图 9.3 中总结了这些方法和它们的参数。

| 绘制字符串、字符和字节的 Graphics 方法                                                                                                                                                                                                                                                                         |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public abstract void drawString {<br/>    String string, // string to be drawn<br/>    int x,         // x coordinate<br/>    int y }        // y coordinate</pre> <p>在坐标(x, y)处使用当前的字体和颜色绘制一个字符串</p>                                                                                       |
| <pre>public void drawChars {<br/>    char chars[], // array to be drawn<br/>    int offset,   // starting subscript(index)<br/>    int number,   // number of element to draw<br/>    int x,        // x coordinate<br/>    int y }       // y coordinate</pre> <p>在坐标(x,y)处使用当前的字体和颜色绘制字符序列</p> |
| <pre>public void drawBytes {<br/>    byte bytes[], // array of bytes<br/>    int offset,   // starting subscript(index)<br/>    int number,   // number of element to draw<br/>    int x,        // x coordinate<br/>    int y }       // y coordinate</pre> <p>在坐标(x,y)处使用当前的字体和颜色绘制字节序列</p>    |

图 9.3 绘制字符串、字符和字节的 Graphics 方法

drawString 方法绘制一个字符串。该方法有 3 个参数——要绘制的 String、x 坐标和 y 坐标。String 使用当前的颜色和字体绘制，当前颜色是指绘制文本和形状所用的颜色，当前字体是绘制文本所用的字体（以下两节要讨论如何设置当前颜色和字体）。点(x,y)对应着字符串的左下角——String 的第一个字符向指定像素坐标的上端和右端进行绘制。因此，如果在坐标(0,0)处绘制一个 String，那么它将是不可见的。

drawChars 方法用于绘制出一系列字符。drawChars 方法有 5 个参数，第一个参数是一个字符数组，第二个参数指明了数组中第一个要绘制字符的下标，第三个参数指明了要绘制的字符个数，其余两个坐标说明了开始绘制的坐标。点(x,y)对应于头一个要绘制字符的左下角。

drawBytes 方法绘制出字节序列。如同 drawChars 方法一样，drawBytes 方法也有五个参数。第一个参数是一个字节数组，第二个参数说明了要绘制的第一个字节在数组中的下标，第三个参数说明了要绘制的元素个数，其余两个参数说明了开始绘制的坐标，点(x,y)对应要绘制字节的左下角。

图 9.4 中的程序使用了 drawString、drawChars 和 drawBytes 方法在 applet 中绘制信息。

```
1 // Fig. 9.4: DrawSCB.java
2 // Demonstrating drawString, drawChars and drawBytes
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class DrawSCB extends Applet {
7     private String s = "Using drawString!";
8     private char c[] = { 'c', 'h', 'a', 'r', 's', ' ', ' ', '8' };
9     private byte b[] = { 'b', 'y', 't', 'e', 1, 2, 3 };
10
11     public void paint( Graphics g )
12     {
13         // draw a string at location (100, 25) on the applet
14         g.drawString( s, 100, 25 );
15
16         // draw a series of characters at location (100, 50)
17         g.drawChars( c, 2, 3, 100, 50 );
18
19         // draw a series of bytes at location (100, 75)
20         g.drawBytes( b, 0, 5, 100, 75 );
21     }
22 }
```

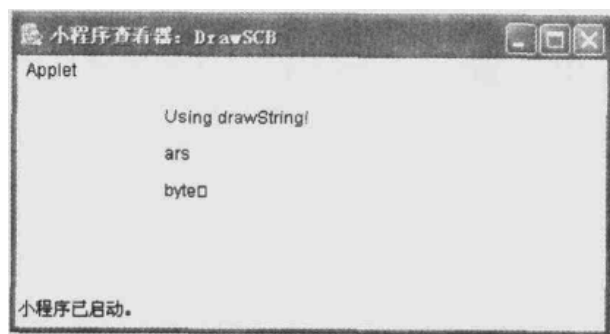


图 9.4 drawString、drawChars 和 drawBytes 方法

在该程序中，drawString 方法在位置(100,25)处显示 “Using drawString!”，下列语句：

```
g.drawChars( c, 2, 3, 100, 50 );
```

在位置(100,50)处显示 “ars”。第二个参数2，指明绘制从字符数组c的下标2（第三个元素）开始。第三个参数3，指明有三个元素将被绘制。方法drawBytes在位置(100,75)显示了一组字节。注意，数组b中头四个元素中的字符作为字符显示，但是第五个元素的数值1却作为一个竖杠显示出来。在这个特定系统中，数值为1的字符不是一个可显示的字符，因此就打印出一个竖杠。

#### 常见编程错误 9.1

向drawChars或drawBytes方法传递一个非法数组下标会抛出一个ArraySubscriptOutOfBoundsException异常。

## 9.4 颜色控制

本节为颜色控制提供了常量和方法。颜色增强了程序的外观特征，用户对程序的印象实际上是通过颜色获取的，颜色传递了丰富的含义。例如，交通灯有3种不同的颜色——红色表明停止，黄

色表明注意，绿色表明通行。

在 `Color` 类中定义了颜色常量和颜色方法，该类继承自 `Object` 类，在图 9.5 中归纳出了预定义的颜色常量。注意，图 9.6 中的两个方法是针对颜色的 `Graphics` 方法。

| 颜色常量                                             | 颜色  | RGB 值         |
|--------------------------------------------------|-----|---------------|
| <code>public final static Color orange</code>    | 橙   | 255, 200, 0   |
| <code>public final static Color pink</code>      | 粉红色 | 255, 175, 175 |
| <code>public final static Color cyan</code>      | 青蓝  | 0, 255, 255   |
| <code>public final static Color magenta</code>   | 品红  | 255, 0, 255   |
| <code>public final static Color yellow</code>    | 黄   | 255, 255, 0   |
| <code>public final static Color black</code>     | 黑   | 0, 0, 0       |
| <code>public final static Color white</code>     | 白   | 255, 255, 255 |
| <code>public final static Color gray</code>      | 灰   | 128, 128, 128 |
| <code>public final static Color lightGray</code> | 浅灰  | 192, 192, 192 |
| <code>public final static Color darkGray</code>  | 深灰  | 64, 64, 64    |
| <code>public final static Color red</code>       | 红   | 255, 0, 0     |
| <code>public final static Color green</code>     | 绿   | 0, 255, 0     |
| <code>public final static Color blue</code>      | 蓝   | 0, 0, 255     |

图 9.5 `Color` 类的 `static` 常量和 RGB 值

RGB 值（红/绿/蓝）创建了每一种颜色。RGB 值由 3 部分组成，所有 3 个 RGB 部分都是范围在 0 到 255 间的整数，或者是范围在 0.0 到 1.0 间的浮点数。第一个 RGB 部分定义了红色的量，第二个 RGB 部分定义了绿色的量，第三个 RGB 部分定义了蓝色的量。RGB 值越大，那种特定颜色的量越大。这样，程序员能够从  $256 \times 256 \times 256$ （或约 16 000 000）种颜色中进行选择。但是，并非所有的计算机都能够显示这么多种颜色。如果有这种情况，那么计算机将显示其所能显示的最相近颜色。

#### 常见编程错误 9.2

使用大写字母拼写 `static Color` 类常量是一种语法错误。

图 9.6 中显示了两个 `Color` 构造函数，一个接收 3 个 `int` 参数，另一个接收 3 个 `float` 参数，每个参数分别表示红色、绿色和蓝色的量。请记住，`int` 值介于 0 到 255 之间而浮点值介于 0.0 到 1.0 之间，因此构造出的 `Color` 具有确定量的红色、绿色和蓝色。`Color` 方法 `getRed`、`getGreen` 和 `getBlue` 返回介于 0 到 255 间的整数，分别代表红色、绿色和蓝色的量。`Graphics` 类的 `getColor` 方法返回一个代表当前绘图颜色的 `Color` 对象。`Graphics` 类的 `setColor` 方法设置当前的绘图颜色。

图 9.7 中的程序改变了当前的颜色并绘制一个 `String`，该程序使用了接收 3 个 `int` 参数的 `Color` 构造函数的版本。

实例变量 `red`、`green` 和 `blue` 已经声明并初始化。在 `paint` 方法中，当前的颜色由下面的语句改变：

```
g.setColor ( new Color ( red , green , blue ) );
```

`setColor` 方法设置由 `red`、`green` 和 `blue` 值构造的 `Color` 对象的当前颜色。接着，`drawString` 方法利用当前新的颜色绘制 `String`。

## Color 方法和与 Graphics 方法有关的颜色

```

public Color (
    int r,    // 0 ~ 255 red content
    int g,    // 0 ~ 255 green content
    int b )   // 0 ~ 255 blue content
    创建一种基于红色、绿色和蓝色量的颜色

public Color (
    float r,  // 0.0 ~ 1.0 red content
    float g,  // 0.0 ~ 1.0 green content
    float b ) // 0.0 ~ 1.0 blue content
    创建一种基于红色、绿色和蓝色量的颜色

public int getRed ()           // Color class
    返回表示红色量的介于 0 到 255 之间的一个值

public int getBlue ()          // Color class
    返回表示蓝色量的介于 0 到 255 之间的一个值

public int getGreen ()         // Color class
    返回表示绿色量的介于 0 到 255 之间的一个值

public abstract Color getColor () // Graphics class
    返回一个表示当前绘图颜色的 Color 对象

public abstract void setColor ( Color c ) // Graphics class
    利用图形颜色设置当前的绘图颜色

```

图 9.6 Color 方法和与 Graphics 方法有关的颜色

```

1  // Fig. 9.7: ShowColors.java
2  // Demonstrating setting and getting a Color.
3  import java.applet.Applet;
4  import java.awt.Graphics;
5  import java.awt.Color;
6
7  public class ShowColors extends Applet {
8      private int red, green, blue;
9
10     public void init()
11     {
12         // set some values
13         red = 100;
14         blue = 255;
15         green = 125;
16     }
17
18     public void paint ( Graphics g )
19     {
20         g.setColor( new Color( red, green, blue ) );
21         g.drawString( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 50, 33 );
22         showStatus( "Current RGB: " + g.getColor().toString() );
23     }
24 }

```

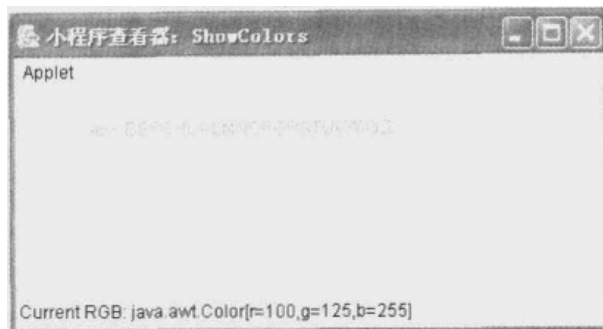


图 9.7 展示设置和获取一个 Color

下列语句:

```
showStatus ( " current RGB : " + g.getColor ( ).toString ( ) );
```

在由getColor返回的Color对象上使用toString方法,结果信息接着在状态栏上显示出来。注意,toString方法将Color对象转换为String表示,其中包含类名和软件包(java.awt.Color),以及红、绿和蓝色值的对象。

图9.8中的程序使用接收3个float参数的Color构造函数来创建一个新的Color对象,改变当前的颜色并使用新的颜色绘制String。

```

1 // Fig. 9.8: ShowColors2.java
2 // Demonstrating the Color constructor with float arguments
3 import java.applet.Applet;
4 import java.awt.Graphics;
5 import java.awt.Color;
6
7 public class ShowColors2 extends Applet {
8     private float red, green, blue;
9
10    public void init()
11    {
12        red = 0.1f;
13        green = 0.21f;
14        blue = 0.33f;
15    }
16
17    public void paint( Graphics g )
18    {
19        g.setColor( new Color( red, green, blue ) );
20        g.drawString( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 60, 33 );
21        showStatus( "Current RGB: " + g.getColor().toString() );
22    }
23 }
```

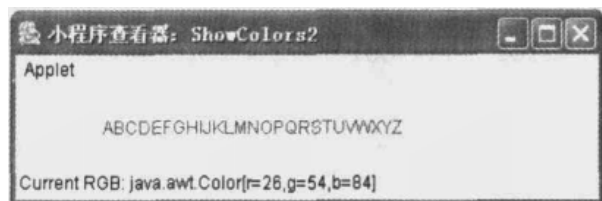


图 9.8 使用 float 参数的 Color 构造函数



实例变量 `red`、`green` 和 `blue` 已经声明和初始化。在 `paint` 方法中可以使用下面的语句改变当前颜色：

```
g.setColor (new Color( red , green , blue ) );
```

`setColor` 方法将当前颜色设置为由 `red`、`green` 和 `blue` 构成的颜色。注意，每个 `red`、`green` 和 `blue` 构成的值用整数值（不是浮点数值）显示在状态栏中。

图 9.9 中的程序将当前颜色设置为 `blue`。在这个程序中，使用下面的语句创建对 `Color` 对象的引用（即实例变量 `c`）：

```
private color c;
```

`c` 的值在 `init` 方法中将设置为常量 `Color.blue`。可以利用下面的语句将当前颜色改成 `blue`：

```
g.setColor ( c );
```

`getRed`、`getGreen` 和 `getBlue` 方法用于获取当前颜色的 RGB 值，RGB 值将显示在状态栏中。

```

1 // Fig. 9.9: ShowColors3.java
2 // Using a predefined static Color object to set the color
3 import java.applet.Applet;
4 import java.awt.Graphics;
5 import java.awt.Color;
6
7 public class ShowColors3 extends Applet {
8     private Color c;
9
10    public void init()
11    {
12        c = Color.blue;
13    }
14
15    public void paint( Graphics g )
16    {
17        g.setColor( c );
18        g.drawString( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 50, 33 );
19        showStatus( "Current RGB: " +
20                    String.valueOf( c.getRed() ) +
21                    " " + String.valueOf( c.getGreen() ) +
22                    " " + String.valueOf( c.getBlue() ) );
23    }
24 }
```

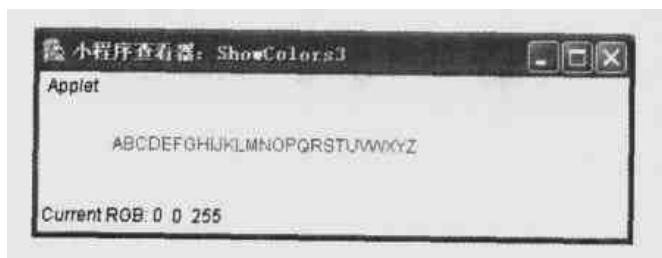


图 9.9 使用一个预定义的 `String Color` 对象来将颜色设置为 `blue`

## 9.5 字体控制

本节介绍字体控制的方法和常量。大多数方法和字体都是Font类的一部分，该类是从Object类继承而来的。这些方法将在图9.10、图9.12和图9.14中给出。

| Font 常量、Font 构造函数和 Graphics 方法 setFont |                                  |
|----------------------------------------|----------------------------------|
| public final static int PLAIN          | //Font class<br>表示普通字体的常量        |
| public final static int BOLD           | //Font class<br>表示粗体的常量          |
| public final static int ITALIC         | //Font class<br>表示斜体的常量          |
| public Font (                          |                                  |
| String s,                              | //font name                      |
| int style,                             | //font style                     |
| int size )                             | //font point size                |
|                                        | 用指定的字体、风格和大小创建 Font 对象           |
| public abstract void setFont ( Font f) | // Graphics class                |
|                                        | 将当前字体设置为由 Font 对象 f 新指定的字体、风格和大小 |

图 9.10 Font 常量、Font 构造函数和 Graphics 方法 setFont

在图9.10中，Font类的static常量PLAIN、BOLD和ITALIC用于说明字体风格。Font构造函数接收3个参数——字体名称、字体风格和字体大小。字体名称为程序运行的系统当前所支持的任意字体，如Courier、Helvetica和TimesRoman。字体风格是Font.PLAIN、Font.ITALIC或Font.BOLD（Font类的static常量）之一，字体风格也可用于组合之中（例如，Font.ITALIC + Font.BOLD）。字体大小用点来度量，1个点是1/72英寸。setFont方法用于设置当前字体，即显示文本所用的字体，setFont以一个Font对象为参数。

### 可移植性提示 9.2

字体的数目随系统的不同而相差较大，JDK保证字体TimesRoman、Courier、Helvetica、Dialog和DialogInput是可用的。

### 软件工程视点 9.1

如果一种字体在系统中不可用，那么Java将使用系统的默认字体。

图9.11中的程序使用3种不同的字体来显示文本，每种字体的大小不同。在程序中使用Font构造函数创建Font对象font1、font2和font3，每次对Font构造函数的调用都将字体名称（TimesRoman、Courier或Helvetica）作为一个String传递，另外还将字体风格（Font.PLAIN、Font.ITALIC或Font.BOLD）和一个字体大小作参数。当前的字体由setFont方法改变。一旦激活了setFont方法，此调用之后的所有文本显示都将使用新的字体。

```

1      // Fig. 9.11: DemoFont.java
2      // Demonstrating the Font constants, the Font constructor
3      // and the setFont method
4      import java.applet.Applet;
5      import java.awt.Graphics;
6      import java.awt.Font;
```

```

7
8   public class DemoFont extends Applet {
9       private Font font1, font2, font3;
10
11       public void init()
12       {
13           // create a font object: 12-point bold times roman
14           font1 = new Font( "TimesRoman", Font.BOLD, 12 );
15
16           // create a font object: 24-point italic courier
17           font2 = new Font( "Courier", Font.ITALIC, 24 );
18
19           // create a font object: 14-point plain helvetica
20           font3 = new Font( "Helvetica", Font.PLAIN, 14 );
21       }
22
23       public void paint( Graphics g )
24       {
25           // set the current font to font1
26           g.setFont( font1 );
27
28           // draw a string in font font1
29           g.drawString( "TimesRoman 12 point bold.", 20, 20 );
30
31           // change the current font to font2
32           g.setFont( font2 );
33
34           // draw a string in font font2
35           g.drawString( "Courier 24 point italic.", 20, 40 );
36
37           // change the current font to font3
38           g.setFont( font3 );
39
40           // draw a string in font font3
41           g.drawString( "Helvetica 14 point plain.", 20, 60 );
42       }
43   }

```

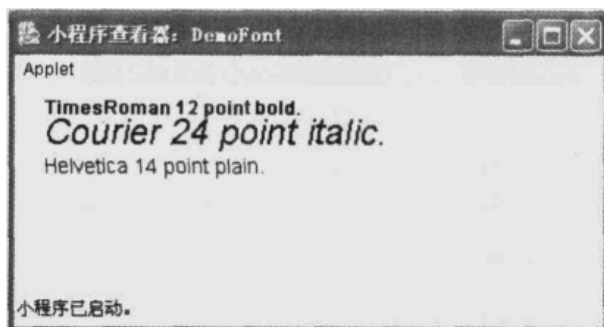


图 9.11 使用 Font 对象和 Graphics 的 setFont 方法设置字体

通常，需要获取当前字体的信息，如字体名称、字体风格和字体大小，在图 9.12 中总结出用于获取字体信息的 Font 方法。getStyle 方法返回一个表示当前类型的整数值，返回的整数值可以是 Font.PLAIN、Font.ITALIC、Font.BOLD，或者为 Font.PLAIN、Font.ITALIC 和 Font.BOLD 的组合之一。

---

**获取字体信息的 Font 方法**

---

```
public int getStyle ()      //Font class
    返回指示当前字体风格的整数值
public int getStyle ()      //Font class
    返回指示当前字体大小的整数值
public String getName ()    //Font class
    以字符串方式返回当前的字体名称
public String getFamily ()  //Font class
    返回作为一个字符串的当前字体族的名称
```

---

图 9.12 获取字体信息的 Font 方法

getSize 方法以点计数来返回字体大小, getName 方法以 String 类型返回当前的字体名称, getFamily 方法返回当前字体所属字体族的名称, 字体族的名称与系统平台有关。

**可移植性提示 9.3**

Java 为实现可移植性而使用标准化的字体名称, 并将它们映射到与系统相关的字体名称上, 这些对程序员都是透明的。

图 9.13 的程序创建了一个 Font 对象 f, 该对象的字体为 24 像素的 Courier 粗斜体。使用 setFont 方法将当前的字体改变为 f。下列语句:

```
style = f.getStyle ();
```

使用 getStyle 来判断当前的字体风格。返回的整数值同 Font 常量相比, 并用一个 String 显示字体的类型。下列语句:

```
size = f.getSize ();
```

判断当前的字体大小。通过对 getName 方法的调用来判断字体名称 (Courier)。字体族 (Courier) 可以通过下面的语句显示出来:

```
g.drawString (" Font family is" + f.getFamily (), 20, 60 );
```

drawString 方法在位置(20,60)处显示文字。请再次注意, getString、getName、getSize 和 getFamily 方法属于 Font 类。

---

```
1 // Fig. 9.13: DemoFont2.java
2 // Demonstrating the Font methods for
3 // retrieving font information
4 import java.applet.Applet;
5 import java.awt.Graphics;
6 import java.awt.Font;
7
8 public class DemoFont2 extends Applet {
9     private Font f;
10
11     public void init()
12     {
13         // create a font object: 24-point bold italic courier
14         f = new Font ("Courier", Font.ITALIC + Font.BOLD, 24 );
15     }
16
17     public void paint( Graphics g )
18     {
```

```

19      int style, size;
20      String s, name;
21
22      g.setFont( f );           // set the current font to f
23      style = f.getStyle();     // determine current font style
24
25      if ( style == Font.PLAIN )
26          s = "Plain ";
27      else if ( style == Font.BOLD )
28          s = "Bold ";
29      else if ( style == Font.ITALIC )
30          s = "Italic ";
31      else // bold + italic
32          s = "Bold italic ";
33
34      size = f.getSize();       // determine current font size
35      s += size + " point ";
36      name = f.getName();       // determine current font name
37      s += name;
38      g.drawString( s, 20, 40 );
39
40      // display font family
41      g.drawString( "Font family is " + f.getFamily(), 20, 60 );
42  }
43  }

```

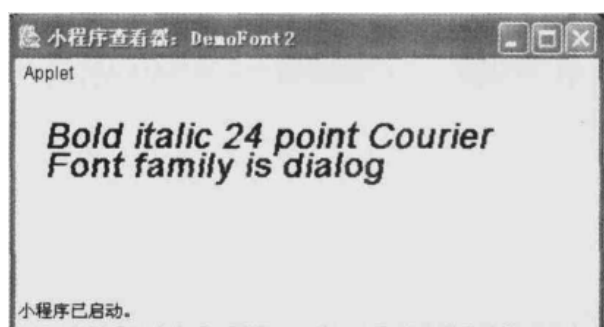


图 9.13 getName、getStyle、getSize 和 getFamily 方法

Font 类的方法中也有用于测试当前字体风格的，它们在图 9.14 中总结出来。如果当前的字体风格为普通类型，则 isPlain 方法返回 true；如果当前字体风格为粗体，则 isBold 方法返回 true；如果当前字体风格为斜体，则 isItalic 方法返回 true。

#### 测试字体风格的 Font 方法

|                                  |               |
|----------------------------------|---------------|
| public boolean isPlain ()        | // Font class |
| 使用普通字体风格测试字体，如果该字体是普通字体，则返回 true |               |
| public boolean isBold ()         | // Font class |
| 使用粗体字体风格测试字体，如果该字体是粗体，则返回 true   |               |
| public boolean isItalic ()       | // Font class |
| 使用斜体字体风格测试字体，如果该字体为斜体，则返回 true   |               |

图 9.14 测试字体风格的 Font 方法

图 9.15 中的程序展示了 isPlain、isBold 和 isItalic 方法的使用。Font 的对象 f 使用下面的语句进行创建：

```
f = new Font ( " Courier ", Font.ITALIC +Font.BOLD , 24);
```

并且使用 `setFont` 方法将其设置为当前字体。`isBold`、`isItalic` 和 `isPlain` 方法用于 `f` 字体风格的 if/else 测试，随后将一个 `String` 绘制在描述该字体风格的 applet 上。

有时候必须了解关于一个字体度量的精确信息——诸如高度 (`height`)、下差 (`descent`，即字符低于基线的部分)、上差 (`ascent`，即字符高于基线的部分)、字冠 (`leading`，即高度去除基线下差和基线上差后剩余的部分) 等。图 9.16 展示了一些常用的字体度量元 (`font metrics`)，注意传递到 `drawString` 内的坐标对应左下角。

在 `FontMetrics` 类中定义了一些获取字体度量元的方法，该类继承自 `Object` 类，在图 9.17 中总结出 `FontMetrics` 方法以及其他来自 `Graphics` 类和 `Toolkit` 类的实用方法。

```
1 // Fig. 9.15: DemoFont3.java
2 // Demonstrating isPlain, isBold and isItalic
3 import java.applet.Applet;
4 import java.awt.Graphics;
5 import java.awt.Font;
6
7 public class DemoFont3 extends Applet {
8     private Font f;
9
10    public void init()
11    {
12        // create a font object: 24-point bold italic courier
13        f = new Font( "Courier", Font.ITALIC + Font.BOLD, 24 );
14    }
15
16    public void paint( Graphics g )
17    {
18        String s;
19
20        g.setFont( f );           // set the current font to f
21
22        if ( f.isPlain() == true )
23            s = "Font is plain.";
24        else if ( f.isBold() == true && f.isItalic() == false )
25            s = "Font is bold.";
26        else if ( f.isItalic() == true && f.isBold() == false )
27            s = "Font is italic.";
28        else // bold + italic
29            s = "Font is bold italic.";
30
31        g.drawString( s, 20, 40 );
32    }
33 }
```

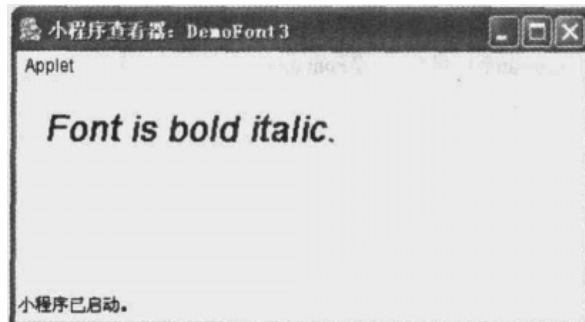


图 9.15 `isPlain`、`isBold` 和 `isItalic` 方法



图 9.16 字体度量元

## 用于获取字体度量元的方法

|                                                                                                         |                                  |
|---------------------------------------------------------------------------------------------------------|----------------------------------|
| <code>public abstract Font getFont ()</code>                                                            | <code>//Graphics class</code>    |
| 返回表示当前字体的 Font 对象                                                                                       |                                  |
| <code>public abstract String[] getFontList ()</code>                                                    | <code>//Toolkit class</code>     |
| 为系统创建一个字体的列表                                                                                            |                                  |
| <code>public int getAscent ()</code>                                                                    | <code>//FontMetrics class</code> |
| 返回用点表示的字体的基线上差值                                                                                         |                                  |
| <code>public int getDescent ()</code>                                                                   | <code>//FontMetrics class</code> |
| 返回用点表示的字体的基线下差值                                                                                         |                                  |
| <code>public int getLeading ()</code>                                                                   | <code>//FontMetrics class</code> |
| 返回用点表示的字体的字冠值                                                                                           |                                  |
| <code>public int getHeight ()</code>                                                                    | <code>//FontMetrics class</code> |
| 返回用点表示的字体的高度值                                                                                           |                                  |
| <code>public int FontMetrics.getFontMetrics ()</code>                                                   | <code>//FontMetrics class</code> |
| 返回当前字体的度量元                                                                                              |                                  |
| 请使用 <code>getAscent</code> 、 <code>getDescent</code> 、 <code>getLeading</code> 和 <code>getHeight</code> |                                  |

图 9.17 来自 Graphics 类、Toolkit 类和 FontMetrics 类的用于获取字体度量元的方法

图 9.18 的程序使用图 9.17 中的方法来获取度量信息，该程序创建了两种字体：一种是 14 像素的 Courier 粗体，另一种是 10 像素的 TimesRoman 字体。下面的语句将对象 font1 设置为当前字体：

```
showStatus ( g.getFont ( ).toString ( ) );
```

```

1 // Fig. 9.18: Metrics.java
2 // Demonstrating methods of the FontMetrics class, Graphics
3 // class and Toolkit class useful for obtaining font metrics
4 import java.applet.Applet;
5 import java.awt.Graphics;
6 import java.awt.Font;
7 import java.awt.Toolkit;
8
9 public class Metrics extends Applet {
10     private Font font1, font2;
11
12     public void init()
13     {
14         font1 = new Font( "Courier", Font.BOLD, 14 );
15         font2 = new Font( "TimesRoman", Font.PLAIN, 10 );
16     }
17
18     public void paint( Graphics g )
19     {
20         g.setFont( font1 ); // set the current font
21     }

```

```

22      // display the current font in the status bar
23      showStatus( g.getFont().toString() );
24
25      // get information about the current font font1
26      int ascent = g.getFontMetrics().getAscent();
27      int descent = g.getFontMetrics().getDescent();
28      int height = g.getFontMetrics().getHeight();
29      int leading = g.getFontMetrics().getLeading();
30
31      String s1 = "Ascent of Font font1 is " +
32                  String.valueOf( ascent );
33
34      String s2 = "Descent of Font font1 is " +
35                  String.valueOf( descent );
36
37      String s3 = "Height of Font font1 is " +
38                  String.valueOf( height );
39
40      String s4 = "Leading of Font font1 is " +
41                  String.valueOf( leading );
42
43      g.drawString( s1, 10, 10 );
44      g.drawString( s2, 10, 20 );
45      g.drawString( s3, 10, 30 );
46      g.drawString( s4, 10, 40 );
47
48      // get information about the font font2
49      ascent = g.getFontMetrics( font2 ).getAscent();
50      descent = g.getFontMetrics( font2 ).getDescent();
51      height = g.getFontMetrics( font2 ).getHeight();
52      leading = g.getFontMetrics( font2 ).getLeading();
53
54      s1 = "Ascent of Font font2 is " +
55          String.valueOf( ascent );
56
57      s2 = "Descent of Font font2 is " +
58          String.valueOf( descent );
59
60      s3 = "Height of Font font2 is " +
61          String.valueOf( height );
62
63      s4 = "Leading of Font font2 is " +
64          String.valueOf( leading );
65
66      g.drawString( s1, 10, 60 );
67      g.drawString( s2, 10, 70 );
68      g.drawString( s3, 10, 80 );
69      g.drawString( s4, 10, 90 );
70
71      g.drawString( " Font list:", 10, 110 );
72
73      // get the list of fonts
74      String fonts[] =
75          Toolkit.getDefaultToolkit().getFontList();
76
77      for ( int i = 0; i < fonts.length; i++ )
78          g.drawString( fonts[ i ], 10, i * 10 + 120 );
79  }
80  ;

```



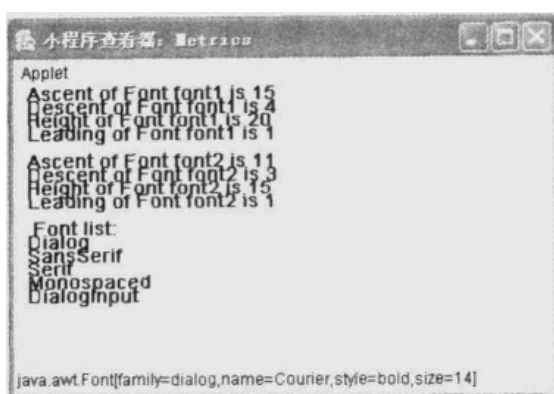


图 9.18 获取字体度量信息和一系列系统所拥有的字体

使用 `toString` 方法将 `getFont` 返回的 `Font` 对象转换成一个 `String`，转换的 `String` 随后显示在状态栏上。`font` 的基线上差、高度、字冠和基线下差由下面 4 行确定：

```
int ascent = g.getFontMetrics ( ).getAscent ( );
int descent = g.getFontMetrics ( ).getDescent ( );
int height = g.getFontMetrics ( ).getHeight ( );
int leading = g.getFontMetrics ( ).getLeading ( );
```

上面几行的字体度量信息将绘制在 applet 上。`font 2` 的基线上差、高度、字冠和基线下差信息以相似的方式汇集在一起并绘制在 applet 中。下列语句：

```
String fonts [] = Toolkit.getDefaultToolkit ( ).getFontList ( );
```

从工具箱 (toolkit) 中获取系统字体并将它们存入一个数组中，工具箱在 Java 和系统间交互作用以获取关于系统的信息。`getDefaultToolkit` 方法是 `Toolkit` 类的一个 `static` 方法 (`Toolkit` 继承自 `Object` 类)。`getDefaultToolkit` 方法获取当前系统的默认工具箱 (即返回一个 `Toolkit` 对象)。`Toolkit` 类的 `getFontList` 方法以 `String` 数组的形式返回系统拥有的字体名称，`for` 循环用于显示系统所拥有的字体。

## 9.6 绘制线条

本节介绍了用于绘制线条的 `Graphics` 类 `drawLine` 方法，在图 9.19 中总结出了 `drawLine` 方法的参数。

`drawLine` 接收 4 个整型参数，头两个参数指明了第一个点的坐标，后两个参数指明了第二个点的坐标，两点间的线段用当前颜色绘制。

| Graphics 类的 drawLine 方法                      |                                           |
|----------------------------------------------|-------------------------------------------|
| <code>public abstract void drawLine (</code> | <code>// Graphics class</code>            |
| <code>int x1,</code>                         | <code>// x coordinate first point</code>  |
| <code>int y1,</code>                         | <code>// y coordinate first point</code>  |
| <code>int x2,</code>                         | <code>// x coordinate second point</code> |
| <code>int y2)</code>                         | <code>// y coordinate second point</code> |
| 使用当前颜色在点(x1,y1)和点(x2,y2)之间画一条线               |                                           |

图 9.19 Graphics 类的 drawLine 方法

图 9.20 的程序使用 `drawLine` 方法在点(10,10)和(230,95)之间绘制了一条线。

```

1 // Fig. 9.20: Line.java
2 // Demonstrating drawLine
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class Line extends Applet {
7
8     public void paint( Graphics g )
9     {
10         // draw a line from (10, 10) to (230, 95)
11         g.drawLine( 10, 10, 230, 95 );
12     }
13 }

```

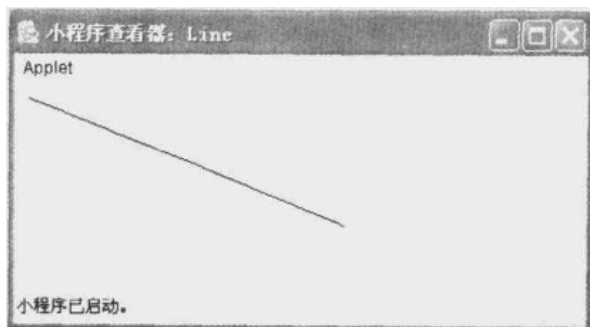


图 9.20 Graphics 类的 drawLine 方法

## 9.7 绘制矩形

本节提供了用于绘制矩形的 Graphics 方法。该方法在图 9.21 中总结出来。

drawRect 方法绘制一个矩形。前两个参数指明了矩形的左上角坐标。第三个参数和第四个参数分别指明了矩形的宽和高。

### 绘制矩形的 Graphics 方法

```

public void drawRect (           // Graphics class
    int x,           // top-left x coordintate
    int y,           // top-left y coordintate
    int width,       // width
    int height )     // height
    绘制指定 width 和 height 的矩形，矩形的左上角坐标为(x1,y1)，使用当前颜色画出该矩形

public void drawRect (           // Graphics class
    int x,           // top-left x coordintate
    int y,           // top-left y coordintate
    int width,       // width
    int height )     // height
    绘制一指定 width 和 height 并用当前颜色填充的矩形，该矩形左上角坐标为(x,y)

public void drawRect (           // Graphics class
    int x,           // top-left x coordintate
    int y,           // top-left y coordintate
    int width,       // width
    int height )     // height
    使用当前背景的颜色绘制一个矩形

```

图 9.21 用于绘制矩形的 Graphics 方法

`fillRect` 方法绘制一个填充的矩形，当前的颜色用于填充该矩形。`fillRect` 方法接收 4 个参数：x 坐标、y 坐标、宽度和高度。点(x,y)对应填充的矩形的左上角。

`clearRect` 方法使用当前的背景色在指定的矩形上绘制一个矩形。该方法接收左上角坐标、矩形的宽度和高度作为参数，以便完成操作。

图 9.22 的程序使用 `drawRect` 和 `fillRect` 方法分别绘制了一个矩形和一个填充矩形。`drawRect` 方法用于在坐标(10,15)上绘制一个矩形。该矩形宽为 100 个像素，高为 100 个像素。`fillRect` 方法用于在坐标(150,15)处绘制一个填充矩形，填充矩形宽为 100 个像素，高为 100 个像素。

```

1 // Fig. 9.22: Rectangle.java
2 // Demonstrating drawRect and fillRect
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class Rectangle extends Applet {
7
8     public void paint( Graphics g )
9     {
10         // draw a rectangle at location ( 10, 15 )
11         g.drawRect( 10, 15, 100, 100 );
12
13         // draw a filled rectangle at location ( 150, 15 )
14         g.fillRect( 150, 15, 100, 100 );
15     }
16 }

```

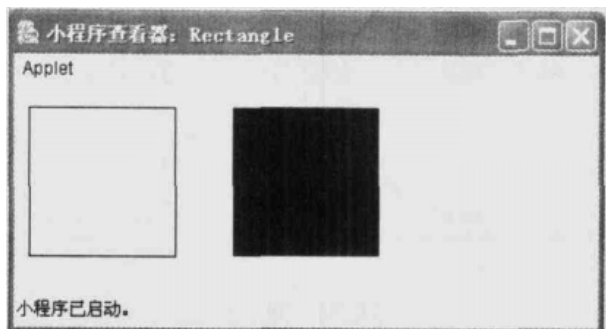


图 9.22 使用 `Graphics` 类的 `drawRect` 方法和 `fillRect` 方法绘制矩形

## 9.8 绘制圆角矩形

本节提供绘制圆角矩形的 `Graphics` 方法，在图 9.23 中总结出了这些方法和它们的参数。

| 绘制圆角矩形的 <code>Graphics</code> 方法                      |                                |
|-------------------------------------------------------|--------------------------------|
| <code>public abstract void drawRoundRect (</code>     | <code>// Graphics class</code> |
| <code>int x,</code>                                   | <code>// x coordinate</code>   |
| <code>int y,</code>                                   | <code>// y coordinate</code>   |
| <code>int width,</code>                               | <code>// width</code>          |
| <code>int height,</code>                              | <code>// height</code>         |
| <code>int arcWidth,</code>                            | <code>// arc width</code>      |
| <code>int arcHeight )</code>                          | <code>// arc height</code>     |
| 以指定的 <code>width</code> 和 <code>height</code> 绘制一圆角矩形 |                                |

(续表)

| 绘制圆角矩形的 Graphics 方法                               |                                |
|---------------------------------------------------|--------------------------------|
| <hr/>                                             |                                |
| <code>public abstract void fillRoundRect (</code> | <code>// Graphics class</code> |
| <code>int x,</code>                               | <code>// x coordinate</code>   |
| <code>int y,</code>                               | <code>// y coordinate</code>   |
| <code>int width,</code>                           | <code>// width</code>          |
| <code>int height,</code>                          | <code>// height</code>         |
| <code>int arcWidth,</code>                        | <code>// arc width</code>      |
| <code>int arcHeight)</code>                       | <code>// arc height</code>     |
| 以指定的 width 和 height 绘制一个使用当前颜色填充的矩形               |                                |
| <hr/>                                             |                                |

图 9.23 用于绘制圆角矩形的 Graphics 类的方法

`drawRoundRect` 方法绘制带有圆角的矩形。前两个参数指明了边界矩形的左上角坐标，边界矩形是绘制圆角矩形的区域。注意，左上角的坐标并非圆角的边缘，而是矩形在非圆角时的边缘。随后两个参数指明了 width 和 height，最后两个参数指明了该矩形的圆角性质，`arcWidth`（弧宽）参数指明了水平分量而 `arcHeight`（弧高）参数指明了垂直分量，使用相同的 `arcWidth` 和 `arcHeight` 值就可以在角上产生四分之一的圆。

图 9.24 标出了弧宽、弧高、宽和高。当 width、height、`arcWidth` 和 `arcHeight` 都具有相同的值时，就产生了一个圆。如果 width 和 height 的值相同而 `arcWidth` 和 `arcHeight` 的值为 0，则将产生一个正方形。

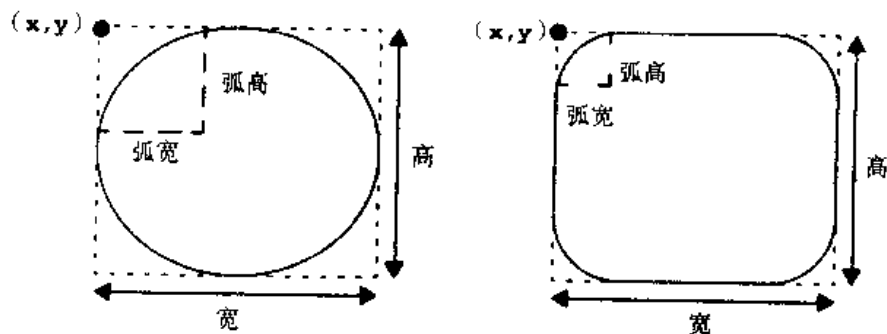


图 9.24 圆角矩形

`fillRoundRect` 方法绘制带有圆角的填充矩形，并使用当前的颜色填充该矩形。`fillRoundRect` 方法接收与 `drawRoundRect` 方法一样的 6 个参数：`x` 坐标、`y` 坐标、宽度、高度、弧宽和弧高，点 `(x,y)` 指明了边界矩形的左上角。

图 9.25 的程序使用 `drawRoundRect` 和 `fillRoundRect` 方法绘制了 3 个圆角矩形和 2 个填充圆角矩形。第一个圆角矩形绘制在 `(10,35)` 处，其宽为 50、高为 50，弧宽和弧高分别为 10 和 20。第二个圆角矩形绘制在 `(80,15)` 处并将其填充，填充矩形的宽和高分别为 60 和 80。第三个圆角矩形构成一个椭圆，该椭圆在 `(150,55)` 进行绘制，其高为 80、宽为 20，弧宽和弧高均为 70。第四个圆角矩形构成了一个填充正方形，该正方形在 `(240,15)` 进行绘制，其宽为 80，高为 80，弧宽和弧高均为 0。最后一个圆角矩形绘制成了一个圆，这个圆绘制在 `(330,15)` 的位置上，其宽、高、弧宽和弧高均为 80。

```

1 // Fig. 9.25: Rectangle2.java
2 // Drawing rounded rectangles
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class Rectangle2 extends Applet {

```

```

7
8     public void paint( Graphics g )
9     {
10         // draw a rounded rectangle at (10, 35)
11         g.drawRoundRect( 10, 35, 50, 50, 10, 20 );
12
13         // draw a filled rounded rectangle at (80, 15)
14         g.fillRoundRect( 80, 15, 60, 80, 50, 10 );
15
16         // draw a rounded rectangle at (150, 55)
17         g.drawRoundRect( 150, 55, 80, 20, 70, 70 );
18
19         // draw a filled square at (240, 15)
20         g.fillRoundRect( 240, 15, 80, 80, 0, 0 );
21
22         // draw a circle at (330, 15)
23         g.drawRoundRect( 330, 15, 80, 80, 80, 80 );
24     }
25 }

```

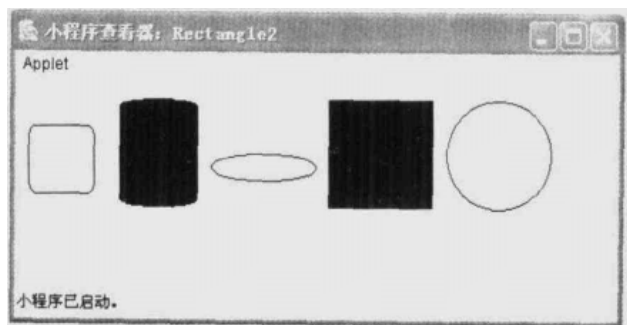


图 9.25 drawRoundRect 方法和 fillRoundRect 方法

## 9.9 绘制三维矩形

本节将介绍用于绘制三维矩形的 Graphics 方法，如图 9.26 中的总结所示。

draw3DRect 和 fill3DRect 接收相同的参数。前两个参数指明了矩形的左上角，随后两个参数指明了矩形的 width 和 height，最后的参数判断矩形是凸起的还是凹陷的。true 值指明矩形是凸起的，而 false 值则指明矩形是凹陷的。三维矩形使用当前颜色进行绘制，某些颜色的应用会使三维效果不太明显。

### 绘制三维矩形的 Graphics 方法

```

public void draw3DRect ( //Graphics class
    int x,                // x coordinate top-left corner
    int y,                // y coordinate top-left corner
    int width,            //width of rectantgle
    int height,           //height of rectantgle
    Boolean b)            //raised when true

```

以指定的 width 和 height 并用当前颜色绘制一个三维矩形，矩形左上角坐标为(x,y)，当 b 为 true 时矩形是凸起的，为 false 时是凹陷的

```

public void draw3DRect ( //Graphics class
    int x,                // x coordinate top-left corner
    int y,                // y coordinate top-left corner

```

(续表)

## 绘制三维矩形的 Graphics 方法

|             |                       |
|-------------|-----------------------|
| int width,  | //width of rectangle  |
| int height, | //height of rectangle |
| Boolean b)  | //raised when true    |

以指定的 width 和 height 绘制三维矩形, 并以当前颜色填充, 该矩形左上角坐标为(x,y), 当 b 为 true 时该矩形是凸起的, 当 b 为 false 时, 该矩形是凹陷的

图 9.26 绘制三维矩形的 Graphics 方法

图 9.27 中的程序展示了 draw3DRect 方法和 fill3DRect 方法。使用黄色绘制三维矩形以增强三维效果, 同时展示了凸起和凹陷的效果。

```

1  // Fig. 9.27: Draw3D.java
2  // Drawing 3-D rectangles
3  import java.applet.Applet;
4  import java.awt.Graphics;
5  import java.awt.Color;
6
7  public class Draw3D extends Applet {
8
9      public void paint( Graphics g )
10     {
11         g.setColor( Color.yellow );
12
13         // draw a raised 3D rectangle at location (10, 10)
14         g.draw3DRect( 10, 10, 100, 100, true );
15
16         // draw a sunk 3D at location (130, 10)
17         g.draw3DRect( 130, 10, 100, 100, false );
18
19         // draw a filled raised 3D rectangle at (10, 120)
20         g.fill3DRect( 10, 120, 100, 100, true );
21
22         // draw a filled sunk 3D rectangle at (130, 120)
23         g.fill3DRect( 130, 120, 100, 100, false );
24     }
25 }

```

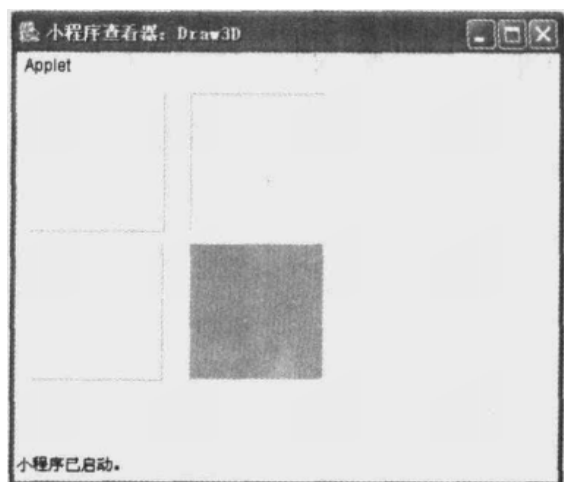


图 9.27 draw3DRect 方法和 fill3DRect 方法

## 9.10 绘制椭圆

本节给出了用于绘制椭圆的 Graphics 方法，如图 9.28 中所示。

| 绘制椭圆的 Graphics 方法                            |                                |
|----------------------------------------------|--------------------------------|
| <code>public abstract void drawOval (</code> | <code>// Graphics class</code> |
| <code>int x,</code>                          | <code>// x coordinate</code>   |
| <code>int y,</code>                          | <code>// y coordinate</code>   |
| <code>int width,</code>                      | <code>// width</code>          |
| <code>int height )</code>                    | <code>// height</code>         |
| 以指定的宽度和高度并用当前颜色绘制椭圆，其外切矩形左上角的坐标为(x,y)        |                                |
| <code>public abstract void fillOval (</code> | <code>// Graphics class</code> |
| <code>int x,</code>                          | <code>// x coordinate</code>   |
| <code>int y,</code>                          | <code>// y coordinate</code>   |
| <code>int width,</code>                      | <code>// width</code>          |
| <code>int height )</code>                    | <code>// height</code>         |
| 以指定的宽度和高度并以当前颜色绘制一个填充椭圆，其外切矩形左上角的坐标为(x,y)    |                                |

图 9.28 绘制椭圆的 Graphics 方法

`drawOval` 方法和 `fillOval` 方法都接收同样的两组参数。头两个参数指明了包含这个椭圆的外切矩形的左上角坐标。后两个参数指明了 `width` 和 `height`。图 9.29 显示了一个矩形中的椭圆。

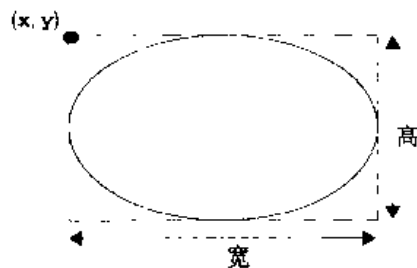


图 9.29 一个矩形内的椭圆

图 9.30 中的程序展示了 `drawOval` 和 `fillOval` 方法，程序在 applet 中绘制了两个椭圆，并且填充了其中的一个。

```

1  // Fig. 9.30: DrawOval.java
2  // Drawing ovals
3  import java.applet.Applet;
4  import java.awt.Graphics;
5
6  public class DrawOval extends Applet {
7
8      public void paint( Graphics g )
9      {
10         // draw an oval at location (10, 15)
11         g.drawOval( 10, 15, 100, 70 );
12
13         // draw an oval at location (160, 15)
14         g.fillOval( 160, 15, 70, 130 );
15     }
16 }
```

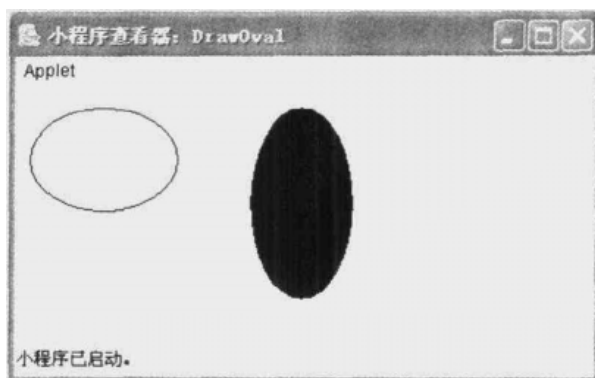


图 9.30 drawOval 方法和 fillOval 方法

## 9.11 绘制圆弧

本节提供了绘制圆弧的 Graphics 方法。一段圆弧是圆的一部分，圆弧的角度通过度数来衡量。一段圆弧在两个角度间绘制——一个为开始角度而另一个为圆弧角度，开始角度为圆弧开始的角度，圆弧角度为圆弧最后的度数。

图 9.31 展示了两段圆弧。左边的坐标系显示圆弧从 0 度画到大约 110 度，以逆时针方向画过的弧用正角度衡量。右边的坐标系显示从 0 度大约画到 -110 度，以顺时针方向画过的圆弧用负角度衡量。在图 9.32 中总结出了在 Graphics 中绘制圆弧的方法。

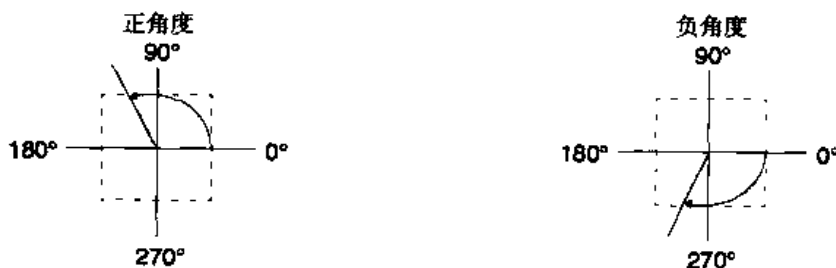


图 9.31 正、负圆弧角度

### 绘制圆弧的 Graphics 方法

```
public abstract void drawArc (           // Graphics class
    int x,                               // x coordinate
    int y,                               // y coordinate
    int width,                           // arc width
    int height,                          // arc height
    int startAngle,                      // beginning angle
    int arcAngle)                       // arc angle
```

在指定宽度和高度的外切矩形内用当前颜色画一段圆弧，外切矩形的左上角坐标为(x,y)，从开始角度画到圆弧的角度

```
public abstract void fillRoundRect ( // Graphics class
    int x,                               // x coordinate
    int y,                               // y coordinate
    int width,                           // arc width
    int height,                          // arc height
```



(续表)

## 绘制圆弧的 Graphics 方法

int startAngle, // beginning angle

int arcAngle) // arc angle

在指定宽度和高度的外切矩形框内用当前的颜色画一段圆弧，外切矩形的左上角坐标为(x,y)，从开始角度画到圆弧的角度

图 9.32 Graphics 类中绘制圆弧的方法

drawArc 方法和 fillArc 方法都接收 6 个参数。前 2 个参数指明了包含圆弧的外切矩形的左上角坐标，第三个参数和第四个参数分别指明了圆弧的宽和高，第五个参数指明了开始角度，最后一个参数指明了圆弧的角度，并用当前颜色绘制圆弧。图 9.33 显示了矩形内接的一段圆弧，从大约 200 度开始画到约 -110 度为止。

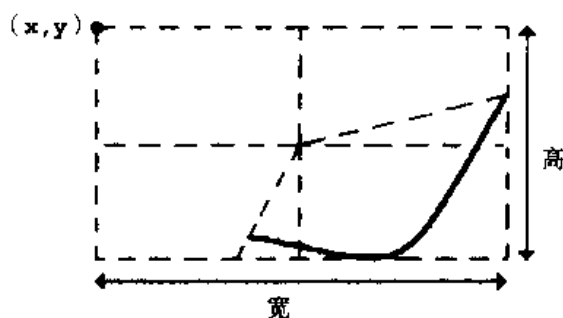


图 9.33 内接于矩形的一段弧

图 9.34 的程序展示了图 9.32 的绘制圆弧的方法，图中绘制了 5 段圆弧，其中有 3 个填充圆弧。

```

1 // Fig. 9.34: DrawArcs.java
2 // Drawing arcs
3 import java.applet.Applet;
4 import java.awt.Graphics;
5
6 public class DrawArcs extends Applet {
7
8     public void paint( Graphics g )
9     {
10         // draw an arc
11         g.drawArc( 15, 15, 80, 80, 0, 180 );
12
13         // draw an arc
14         g.drawArc( 100, 100, 80, 80, 0, 110 );
15
16         // draw a solid arc
17         g.fillArc( 100, 15, 70, 80, 0, 270 );
18
19         // draw a solid arc
20         g.fillArc( 15, 70, 70, 80, 0, -110 );
21
22         // draw a solid arc
23         g.fillArc( 190, 15, 80, 140, 0, -360 );
24     }
25 }

```



图 9.34 drawArc 方法和 fillArc 方法

## 9.12 绘制多边形

多边形 (polygon) 是含有多条边的形状。多边形使用图 9.35 中的 Graphics 方法进行绘制, 注意, 其中一些方法需要 Polygon 对象——它由 Polygon 类创建。在图 9.35 中也描述了 Polygon 类的构造函数。

图 9.35 中的第一个 drawPolygon 方法接收 3 个参数: 一个包含 x 坐标的整型数组, 一个包含 y 坐标的整型数组, 还有多边形的顶点数目。如果未将最后一个顶点指定为与第一个顶点相同, 则将会产生一个开放多边形 (open polygon) ——最后一个顶点未同第一个顶点相连。

图 9.35 中的第二个 drawPolygon 方法需要将一个 Polygon 对象作为参数, Polygon 类继承自 Object 类。使用 fillPolygon 方法即可绘制实心多边形 (solid polygon), 也可以绘制填充多边形 (filled polygon)。图 9.35 中的第一个 fillPolygon 方法接收 3 个参数: 一个包含 x 坐标的整数数组, 一个包含 y 坐标的整数数组, 还有多边形中顶点的数目, 并使用当前的颜色填充此多边形。

### 绘制多边形的 Graphics 方法和 polygon 构造函数

```
public abstract void drawPolygon(           // Graphics class
    int xPoints[],           // x coordinate
    int yPoints[],           // y coordinate
    int Points)              // number of points
    使用当前颜色绘制 xPoints、yPoints 数组指定的多边形, 每个顶点的 x 坐标由 xPoints 数组指定, 每个点的 y 坐标由 yPoints 数组指定

public void drawPolygon( polygon p )        // Graphics class
    使用当前颜色绘制一个多边形

public abstract void drawPolygon(           // Graphics class
    int xPoints[],           // x coordinate
    int yPoints[],           // y coordinate
    int Points)              // number of points
    使用当前颜色绘制一个 xPoints、yPoints 数组指定的填充多边形, 每个点的 x 坐标由 xPoints 数组指定, 其 y 坐标由 yPoints 数组指定

public void fillPolygon( polygon p )        // Graphics class
    绘制一个以当前颜色填充的多边形

public polygon()                        // polygon class
    构造一个新的多边形对象, 该多边形不包含任何顶点

public polygon (                        // polygon class
    int xValues[],           // x coordinate
```

(续表)

## 绘制多边形的 Graphics 方法和 polygon 构造函数

```
int yValues[],          // y coordinate
int numberOfpoints)      // number of points
创建一个新多边形对象,该多边形具有 numberOfpoints 条边,各点的 x 坐标来自于 xValues, y 坐标来自于 yValues
```

图 9.35 用于绘制多边形的 Graphics 方法以及 Polygon 构造函数

图 9.35 中的第二个 fillPolygon 方法需要一个 Polygon 对象。填充多边形通常为闭合多边形 (closed polygon), 并且最后一个顶点不必与第一个顶点重合 (即最后一个顶点自动连接第一个顶点)。

## 常见编程错误 9.3

在 drawPolygon 方法或 fillPolygon 方法中, 第三个参数指明的顶点数目如果小于用于显示的多边形坐标数组中元素的个数, 则会产生逻辑错误。

## 常见编程错误 9.4

在 drawPolygon 方法或 fillPolygon 方法中, 第三个参数指明的顶点数目如果大于用于显示的多边形坐标数组中元素的个数, 则会抛出一个 ArrayIndexOutOfBoundsException 异常。

图 9.36 的程序绘制了 5 个多边形, 其中 3 个是填充多边形, 并且使用了 9.35 中的每种方法和构造函数。

```
1 // Fig. 9.36: DrawPoly.java
2 // Drawing polygons
3 import java.applet.Applet;
4 import java.awt.Graphics;
5 import java.awt.Polygon;
6
7 public class DrawPoly extends Applet {
8     // coordinates for first polygon
9     private int xValues[] = { 20, 40, 50, 30, 20, 15, 20 };
10    private int yValues[] = { 20, 20, 30, 50, 50, 30, 20 };
11
12    // coordinates for second polygon
13    private int xValues2[] = { 70, 90, 100, 80, 70, 65,
14                               60, 70 };
15
16    private int yValues2[] = { 70, 70, 80, 100, 100, 80,
17                               60, 70 };
18
19    // coordinates for third polygon
20    private int xValues3[] = { 120, 140, 150, 190 };
21    private int yValues3[] = { 10, 40, 50, 30 };
22
23    // create references to polygons
24    private Polygon p4, p5, p6;
25
26    public void init()
27    {
28        // instantiate polygon objects
29        p4 = new Polygon( xValues, yValues, 7 );
30        p5 = new Polygon();
31        p6 = new Polygon();
32
33        // add points to p5
```

```
34      p5.addPoint( 165, 105 );
35      p5.addPoint( 175, 120 );
36      p5.addPoint( 270, 170 );
37      p5.addPoint( 200, 190 );
38      p5.addPoint( 130, 150 );
39      p5.addPoint( 165, 105 );
40
41      // add points to p6
42      p6.addPoint( 240, 50 );
43      p6.addPoint( 260, 70 );
44      p6.addPoint( 250, 90 );
45  }
46
47  public void paint( Graphics g )
48  {
49      // draw a polygon of 8 points
50      g.drawPolygon( xValues2, yValues2, 8 );
51
52      // draw a polygon object
53      g.drawPolygon( p4 );
54
55      // draw a filled polygon object
56      g.fillPolygon( p5 );
57
58      // draw a filled polygon object
59      g.fillPolygon( p6 );
60
61      // draw a filled polygon of 4 points
62      g.fillPolygon( xValues3, yValues3, 4 );
63  }
64  }
```

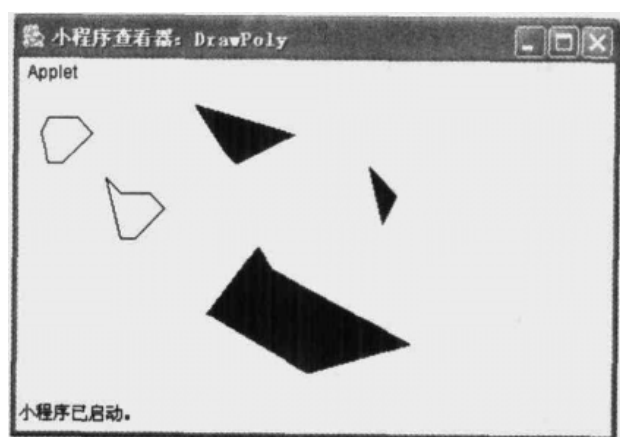


图 9.36 drawPolygon 方法和 fillPolygon 方法

这里声明和初始化了 6 个整型数组。xValues、xValues2 和 xValues3 数组代表 3 个多边形的 x 坐标，yValues、yValues2、yValues3 数组代表 3 个多边形的 y 坐标。

Polygon 类用于创建 3 个对象。下列语句：

```
private Polygon p4, p5, p6;
```

创建了 Polygon 的引用 p4、p5 和 p6。下列语句：

```
p4 = new Polygon ( xValues , yValues, 7 );
```

```
p5 = new Polygon ( );
p6 = new Polygon ( );
```

构造了 Polygon 对象并将它们赋给引用 p4、p5 和 p6。使用 Polygon 方法 addPoint 向 p5 和 p6 添加顶点数目。addPoint 方法接收两个参数，一个为 x 坐标，一个为 y 坐标。下列语句：

```
g.drawPolygon ( xValues2, yValues2, 8 );
```

绘制了第 1 个多边形。下列语句：

```
g.drawPolygon ( p4 );
```

绘制了第 2 个多边形。下列语句：

```
g.fillPolygon ( p5 );
g.fillPolygon ( p6 );
g.fillPolygon ( xValues3, yValues3, 4 );
```

分别绘制了第 3 个、第 4 个和第 5 个多边形。

## 9.13 屏幕操作

本节将介绍用于屏幕操作的 Graphics 方法——copyArea。copyArea 方法复制一块屏幕矩形区域，并将副本放在屏幕的另一位置上，其使用方法如图 9.37 所示。

copyArea 方法接收 6 个参数。前两个参数指明要复制区域的左上角，第 3 个参数和第 4 个参数分别指明了要复制区域的 width 和 height，第 5 个参数和第 6 个参数分别指明要放置副本的位置相对前两个值的偏移量。

| Graphics 方法 copyArea                                                               |                           |
|------------------------------------------------------------------------------------|---------------------------|
| public abstract void copyArea (                                                    | // Graphics class         |
| int x,                                                                             | // top-left x coordinate  |
| int y,                                                                             | // top-left y coordinate  |
| int width,                                                                         | // width of area to copy  |
| int height,                                                                        | // height of area to copy |
| int dx,                                                                            | // horizontal distance    |
| int dy)                                                                            | // vertical distance      |
| 复制指定 width 和 height 的屏幕区域。坐标(x,y)是要复制区域的左上角，dx 和 dy 的值定义了从(x,y)开始的偏移量，副本相对于(x,y)放置 |                           |

图 9.37 用于复制屏幕区域的 Graphics 方法

图 9.38 中的程序使用 copyArea 方法将一个矩形区域 (100 × 100) 复制到点(140,10)。

该程序绘制了两个多边形。一个矩形区域为 100 × 100，起始点为(0,0)，然后使用下列语句复制到位置(140,10)：

```
g.copyArea ( 0, 0, 100, 100, 140, 10 );
```

将水平偏移 140 加到 x 坐标 0 上，并将垂直偏移 10 加到 y 坐标 0 上。

```
1 // Fig. 9.38: DemoCopyArea.java
2 // Demonstrate copying one area of the screen to another
```

```

3    // area of the screen
4    import java.applet.Applet;
5    import java.awt.Graphics;
6
7    public class DemoCopyArea extends Applet {
8
9        // coordinates for first polygon
10       int xValues[] = { 20, 40, 50, 30, 20, 15, 20 };
11       int yValues[] = { 20, 20, 30, 50, 50, 30, 20 };
12
13       // coordinates for second polygon
14       int xValues2[] = { 70, 90, 100, 80, 70, 65, 60, 70 };
15       int yValues2[] = { 70, 70, 80, 100, 100, 80, 60, 70 };
16
17       public void paint( Graphics g )
18       {
19           // draw a polygon of 7 points
20           g.drawPolygon( xValues, yValues, 7 );
21
22           // draw a filled polygon of 8 points
23           g.fillPolygon( xValues2, yValues2, 8 );
24
25           // copy 100 × 100 area of applet to (140, 10)
26           g.copyArea( 0, 0, 100, 100, 140, 10 );
27       }
28   }

```

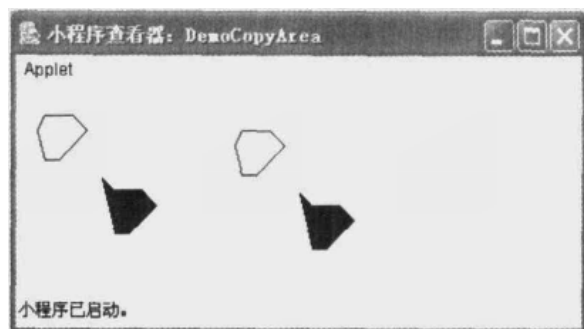


图 9.38 copyArea 方法

## 9.14 绘图模式

绘图模式描述了图形是如何绘制的。默认的绘图模式为覆盖绘图模式 (overwrite paint mode), 即将一个图形绘制在另一个图形的上面。当绘制一个形状时, 该形状就覆盖了一切内容。另外还有 XOR 绘图模式 (XOR paint mode), 它允许所有覆盖的形状都可见。XOR 绘图模式由 Graphics 类的 setXORMode 方法进行设置。setXORMode 方法的开头为:

```
public abstract void setXORMode ( Color c )
```

setXORMode 方法以一个 Color 对象为参数, 这个颜色称为 XORMode 颜色。当覆盖的对象使用相同的颜色绘制时, 则覆盖部分将使用 XORMode 颜色进行绘制。

图 9.39 中的程序给出了绘制 6 种形状的两种绘图模式, 每种形状都与另一种形状重叠。首先绘制一个品红色椭圆; 然后绘制一个黄色矩形, 该黄色矩形利用覆盖绘图模式进行绘制, 并且遮盖了

品红色椭圆的一部分。接着再绘制一个橙色矩形，该橙色矩形利用覆盖绘图模式进行绘制，并且覆盖了黄色矩形的一部分。然后使用下面的语句设置 XOR 模式：

```
g.setXORMode ( Color.yellow );
```

XORMode 颜色是黄色。接着绘制一个使用 XOR 模式的椭圆，并且使用当前的颜色。在橙色矩形上绘制橙色椭圆时将使用 XORMode 颜色，即使用黄色绘制椭圆。黄色矩形覆盖椭圆的一部分用橙色绘制。最后，在黄色矩形上绘制一个蓝色的圆（一个 360 度圆弧）和一个红色正方形。

```

1  // Fig. 9.39: PaintMode.java
2  // Demonstrating the XOR paint mode
3  import java.applet.Applet;
4  import java.awt.Graphics;
5  import java.awt.Color;
6
7  public class PaintMode extends Applet {
8
9      public void paint( Graphics g )
10     {
11         // draw pink oval
12         g.setColor( Color.pink );
13         g.fillOval( 20, 10, 100, 50 );
14
15         // draw a yellow rectangle over part of the oval
16         g.setColor( Color.yellow );
17         g.fillRect( 100, 10, 100, 50 );
18
19         // draw an orange rectangle
20         g.setColor( Color.orange );
21         g.fillRect( 190, 10, 80, 50 );
22
23         // set XOR mode to yellow
24         g.setXORMode( Color.yellow );
25         g.fillOval( 180, 25, 60, 20 );
26
27         // draw a blue arc
28         g.setColor( Color.blue );
29         g.fillArc( 150, 20, 20, 20, 0, 360 );
30
31         // draw a red square
32         g.setColor( Color.red );
33         g.fillRect( 120, 25, 20, 20 );
34     }
35 }
```

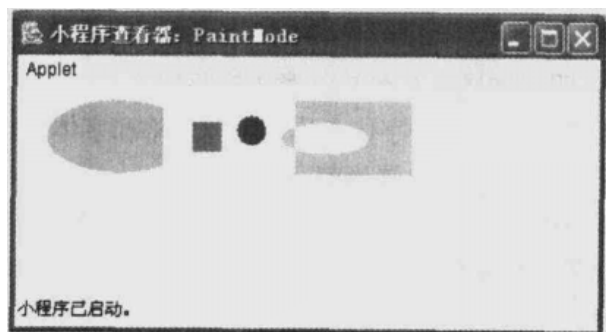


图 9.39 setXORMode 方法

## 小结

- java.awt 软件包中包含进行图形操作所必需的类。
- Color 类包含用于操作颜色的方法和常量。
- Font 类包含操作字体的方法和常量。
- FontMetrics 类包含获取字体信息的方法。
- Polygon 类包含创建多边形的方法。
- Graphics 类包含绘制各种形状的方法。
- Toolkit 类包含 AWT 使用的工具。
- 一个坐标描述了屏幕上的每一点，屏幕的左上角坐标为(0,0)。坐标由一个 x 坐标（距左上角的水平距离）和一个 y 坐标（距左上角的垂直距离）组成。
- x 轴描述了每个 x 坐标而 y 轴描述了每个 y 坐标，文本和形状都可利用坐标精确地绘出。单位用像素来度量，像素是监视器的最小分辨率单位。
- 图形环境具有绘图功能。Graphics 对象通过控制如何绘制信息来管理图形环境。Graphics 对象包括绘图、字体操作和颜色操作等方法。
- Graphics 类是一个 abstract 类——不能实例化 Graphics 类。
- repaint 方法调用 Component 类的 update 方法，后者又调用 paint 方法。程序员通常频繁调用 repaint 方法来强制调用 paint，并且有时为了“平滑”动画（即减少“跳动”）而重写 update 方法。
- Graphics 方法 drawString 接收 3 个参数——要绘制的 String、x 坐标和 y 坐标。使用当前的颜色和字体来绘制 String，点(x,y)对应 String 的左下角。
- Graphics 方法 drawChars 接收 5 个参数——字符数组、数组的开始下标、要绘制的字符个数和显示字符的坐标，点(x,y)对应绘制的字符的左下角。
- Graphics 方法 drawBytes 接收 5 个参数。除了第一个参数是传递进来的 byte 数组之外，其他参数同方法 drawChars 的一样。
- 颜色常量和颜色方法在 Color 类中定义。每种颜色都由 RGB 值（红/绿/蓝）创建，RGB 值由 3 个整型或 3 个浮点部分组成。每个 RGB 部分是一个 0 到 255 的整数，或是 0.0 到 1.0 间的浮点数。RGB 的第一部分定义了红色的量，第二部分定义了绿色的量，第三部分定义了蓝色的量。RGB 值越大，则某种颜色的量越大。
- Color 类的 getRed 方法、getGreen 方法和 getBlue 方法用于返回 0 到 255 间的整数，分别代表红、绿和蓝量值。Graphics 类的 getColor 方法用于返回一个代表当前颜色的 Color 对象，Graphics 类的 setColor 方法用于设置当前颜色。
- 大多数字体方法和常量都是 Font 类的一部分，该类继承自 Object 类。static 常量 PLAIN、BOLD 和 ITALIC，用于指明字体风格并在 Font 类中进行定义。
- Font 构造函数接收 3 个参数——字体名称、字体风格和字体大小。
- Graphics 类的 setFont 方法用于设置当前字体。
- Font 类的 getStyle 方法用于返回一个代表当前类型的整数值。返回的整数值可以是 Font.PLAIN、Font.ITALIC、Font.BOLD，也可以是 Font.PLAIN、Font.ITALIC 和 Font.BOLD 的组合之一。
- Font 类的 getSize 方法用于返回字体大小，返回的整数值代表以点数计算的字体大小。Font 类的 getName 方法用于以一个 String 形式返回当前的字体名称。



- Font类的getFamily方法用于返回当前字体所属的字体族名称,字体族名称与系统平台有关。
- Font类的isPlain方法用于在当前字体风格为普通类型时返回true, isBOLD方法用于在当前字体风格为粗体时返回true, isITALIC方法用于在当前字体为斜体时返回true。
- 字体度量元包括高度、基线下差(字符低于基线的量)、基线上差(字符高于基线的量)和字冠(高度去除基线下差和基线上差后的剩余部分)。
- 在FontMetrics类中定义了获取字体度量元的方法。
- 工具箱在Java和系统间交互作用,以获取关于系统的信息。getDefaultToolkit方法是Toolkit类的static方法,该方法用于获取当前系统的默认工具箱(即返回一个Toolkit对象)。Toolkit类的getFontList方法用于返回系统所拥有的字体的名称。
- Graphics类的drawLine方法使用当前颜色在两个指定的点之间绘制一条线。
- Graphics类的drawRect方法使用指定的左上角坐标、宽度和高度绘制一个矩形。
- Graphics类的fillRect方法用指定的左上角坐标、宽度和高度绘制一个填充的矩形。当前的颜色用于填充矩形。
- Graphics类的clearRect方法使用当前背景色在指定的矩形上绘制一个矩形。
- Graphics类的drawRoundRect方法使用指定的左上角坐标、宽度、高度、弧宽和弧高绘制一个圆角矩形。
- Graphics类的fillRoundRect方法使用指定的左上角坐标、宽度、高度、弧宽和弧高绘制一个填充的圆角矩形。当前的颜色用于填充矩形。
- Graphics类的draw3DRect方法用于绘制一个三维矩形, fill3DRect方法用于绘制一个填充的三维矩形。draw3DRect和fill3DRect接收同样的参数,其中最后一个参数决定该矩形是凸起的还是凹陷的。true值指明该矩形是凸起的,而false值则指明该矩形是凹陷的。
- 圆弧是圆的一部分。圆弧角度在Java中用度数来度量,圆弧是在两个角度之间绘制的,即开始角度和圆弧角度。开始角度是圆弧开始的角度,而圆弧角度是该圆弧的最终角度。
- 以逆时针方向画出的圆弧用正度数度量,以顺时针方向画出的圆弧用负度数度量。
- Graphics类的drawArc方法用于绘制一段圆弧,而fillArc方法用于绘制一段填充圆弧。drawArc方法和fillArc方法都接收6个参数——左上角的x坐标和y坐标、弧宽、弧高、开始角度以及圆弧角度。
- 多边形是含有多条边的图形。Polygon对象由Polygon类创建,Graphics类的drawPolygon方法可以接收3个参数:一个包含x坐标的整数数组,一个包含y坐标的整数数组,还有多边形的顶点数目。如果定义的最后—个顶点不同于第一个顶点,则将会产生一个开放多边形,即最后一个顶点未同第一个顶点相连。另外,drawPolygon方法也可以接收一个Polygon对象。
- Graphics类的fillPolygon方法用于绘制实心多边形或填充多边形,填充多边形总是闭合多边形——最后一个顶点总是会与第一个顶点相连。
- Polygon类的addPoint方法用于向多边形添加一个顶点并接收两个参数,即x坐标和y坐标。
- Graphics类的copyArea方法用于复制屏幕的一个矩形区域,并在屏幕的另一个区域绘制副本的区域。
- 绘图模式描述了如何绘制图形。默认的绘图模式是覆盖绘图模式,即绘制一个形状后,该形状就会覆盖其下面的一切内容。另一种绘图模式为XOR绘图模式,它允许覆盖的形状可见。可以使用Graphics类的setXORMode方法设置XOR绘图模式,该方法以一个Color对象为参数。

## 术语

- addPoint method addPoint 方法  
angle 角度  
arc bounded by a rectangle 矩形内接的弧  
arc height 弧高  
arc sweeping through an angle 经过一个角度的圆弧  
arc width 弧宽  
ascent 基线上差  
awt hierarchy awt 层次  
awt package awt 软件包  
background color 背景颜色  
baseline 基线  
bounding rectangle 外切矩形  
clearRect method clearRect 方法  
closed polygon 闭合多边形  
Color.balek  
Color.blue  
Color.cyan  
Color.darkGray  
Color.gray  
Color.green  
Color.lightGray  
Color.magenta  
Color.orange  
Color.pink  
Color.red  
Color.white  
Color.yellow  
Color class Color 类  
Color constructor Color 构造函数  
Color object Color 对象  
Component class Component 类  
coordinate 坐标  
coordinate system 坐标系  
copyArea method copyArea 方法  
Courier font Courier 字体  
current color 当前颜色  
current font 当前字体  
degree 度数  
descent 基线下差  
draw3DRect method draw3DRect 方法  
drawArc method drawArc 方法  
drawBytes method drawBytes 方法  
drawChars method drawChars 方法  
drawing arc 绘制圆弧  
drawing bytes 绘制字节  
drawing strings 绘制字符串  
drawLine method drawLine 方法  
drawOval method drawOval 方法  
drawRect method drawRect 方法  
drawPolygon method drawPolygon 方法  
drawRoundRect method drawRoundRect 方法  
drawString method drawString 方法  
event 事件  
event-driven process 事件驱动过程  
fill3DRect method fill3DRect 方法  
fillArc method fillArc 方法  
filled polygon 填充多边形  
fillOval method fillOval 方法  
fillPolygon method fillPolygon 方法  
fillRect method fillRect 方法  
fillRoundRect method fillRoundRect 方法  
Font.BOLD  
Font.ITALIC  
Font.PLAIN  
font 字体  
Font class Font 类  
Font constructor Font 构造函数  
font metrics 字体度量元  
font name 字体名称  
font style 字体风格  
FontMetrics class FontMetrics 类  
getAscent method getAscent 方法  
getBlue method getBlue 方法  
getDescent method getDescent 方法  
getGreen method getGreen 方法  
getFamily method getFamily 方法

|                       |                   |                     |               |
|-----------------------|-------------------|---------------------|---------------|
| getFont method        | getFont 方法        | paint mode          | 绘图模式          |
| getFontList method    | getFontList 方法    | pixel               | 像素            |
| getFontMetrics method | getFontMetrics 方法 | point               | 点             |
| getHeight method      | getHeight 方法      | polygon             | 多边形           |
| getLeading method     | getLeading 方法     | Polygon class       | Polygon 类     |
| getName method        | getName 方法        | Polygon constructor | Polygon 构造函数  |
| getRed method         | getRed 方法         | positive degrees    | 正角度           |
| getSize method        | getSize 方法        | repaint method      | repaint 方法    |
| getStyle method       | getStyle 方法       | RGB value           | RGB 值         |
| Graphics class        | Graphics 类        | setColor method     | setColor 方法   |
| graphics context      | 图形环境              | setFont method      | setFont 方法    |
| graphics object       | 图形对象              | setXORMode method   | setXORMode 方法 |
| height                | 高度                | solid polygons      | 实心多边形         |
| Helvetica font        | Helvetica 字体      | TimesRoman font     | TimesRoman 字体 |
| horizontal component  | 水平分量              | toolkit             | 工具箱           |
| isBold method         | isBold 方法         | Toolkit class       | Toolkit 类     |
| isItalic method       | isItalic 方法       | update method       | update 方法     |
| isPlain method        | isPlain 方法        | vertical component  | 垂直分量          |
| leading               | 字冠                | x-axis              | x 轴           |
| negative degrees      | 负角度               | x-coordinate        | x 坐标          |
| open polygon          | 开放多边形             | XOR paint mode      | XOR 绘图模式      |
| overwrite paint mode  | 覆盖绘图模式            | y-axis              | y 轴           |
| Paint interface       | Paint 接口          | y-coordinate        | y 坐标          |

## 自测练习

### 9.1 填空:

- setName 的返回类型是\_\_\_\_\_。
- 可以使用\_\_\_\_\_方法复制一个屏幕的矩形区域。
- \_\_\_\_\_方法用于在两点间绘制一条线。
- RGB 是\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_的缩写。
- 字体大小用\_\_\_\_\_作为单位度量。
- \_\_\_\_\_方法用于绘制一系列字节。

### 9.2 判断下列问题是否正确。如果不正确,请解释原因。

- drawOval(x,y,50,100) 的头两个参数指明了椭圆的中心坐标。
- 在 Java 坐标系中, x 的值从左到右增长。
- fillPolygon 方法使用当前的颜色绘制实心多边形。
- drawArc 方法允许负角度。
- drawRoundRect 方法可用于绘制一个圆。
- getSize 以厘米为单位返回当前字体的大小。

g) 像素坐标(0,0)定位在监视器的正中心。

9.3 找出下列语句中的错误并解释如何改正。这里假定 g 是一个 Graphics 对象。

- a) `Char x [ ] = { '1', '2', '0', '2', 'x', '*', 'p', 'c', 'm' };`  
`g.drawChars ( x, 6, 8, 435, 80 );` //draw chars PCM
- b) `g.setFont ( "Helvetica" );`
- c) `g.erase ( x, y, w, h );` //clear rectangle at (x,y)
- d) `Font fish = new Font( "Carp", Font.BOLDITALIC, 12 );`
- e) `g.setColor ( Color.Yellow );` //change color to yellow

## 自测练习答案

- 9.1 a)String。 b)copyArea。 c)drawLine。 d)红，绿，蓝。 e)点数。 f)drawBytes。
- 9.2 a)不正确。前两个参数指明矩形的左上角。  
 b)正确。  
 c)正确。  
 d)正确。  
 e)正确。  
 f)不正确。字体大小用点数度量。  
 g)不正确。坐标(0,0)对应监视器的左上角。
- 9.3 a) 最后一个合法下标越界。drawChars 方法将试图绘制 8 个字符——从第 6 个下标开始，8 应改为 3。  
 b) setFont 方法以一个 Font 对象为参数，而不是一个 String。  
 c) Graphics 类没有 erase 方法。应该使用 clearRect 方法。  
 d) Carp 不是一个合法字体。应该使用诸如 Courier、TimesRoman 等字体。  
 e) “Yellow” 应以小写字母开头：g.setColor ( Color.yellow );

## 练习

- 9.4 填空：
  - a) getSize 方法属于 \_\_\_\_\_ 类。
  - b) isBold 的返回类型为 \_\_\_\_\_。
  - c) 说明字体风格的 3 个常量是 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
  - d) \_\_\_\_\_ 方法返回表示 RGB 绿色值的整数以代表某种颜色。
- 9.5 判断下列语句是否正确。如果不正确，请解释原因。
  - a) drawPolygon 方法自动连接多边形的最后顶点。
  - b) drawLine 方法在两点之间绘制一条线。
  - c) fillArc 方法使用度数来说明角度。
  - d) 在 Java 坐标系内，y 值从上到下增加。
  - e) Graphics 类是一个 abstract 类。

- f) Font 类直接从 Graphics 类继承。
- 9.6 找出下列语句中的错误,并解释如何改正。
- a) `String ListOfFonts[ ] = Toolkit.getFontList ( );`
  - b) `Toolkit.setXORMode ( green );`
  - c) `String x = graphicsObject.getAscent ( );`
  - d) `Graphics myGraphics = new Graphics ( );`
- 9.7 编写一个程序,使用 `drawRoundRect` 方法绘制 8 个同心圆,各圆应相差 10 个像素。
- 9.8 编写一个程序,使用 `drawArc` 方法绘制 8 个同心圆,各圆应相差 10 个像素。
- 9.9 编写一个程序,使用 `drawArc` 方法绘制一个螺旋线。
- 9.10 编写一个程序,使用 `fillArc` 方法绘制一个螺旋线。
- 9.11 编写一个程序,绘制几条随机长度的线条。
- 9.12 编写一个程序,绘制几条随机颜色的线条。
- 9.13 编写一个程序,综合练习 9.11 和练习 9.12 中要求实现的功能。
- 9.14 编写一个程序,获取系统中的字符并使用每种字体绘制一个 12 点的粗体字符串。
- 9.15 编写一个程序,显示不同大小的 5 个三角形,每个三角形应填充不同的颜色。
- 9.16 编写一个程序,使用不同的字体大小随机绘制字符。
- 9.17 编写一个程序,使用不同的颜色随机绘制字符。
- 9.18 编写一个程序,使用 `drawLine` 方法绘制  $8 \times 8$  方格。
- 9.19 编写一个程序,使用 `drawRect` 方法绘制  $10 \times 10$  方格。
- 9.20 编写一个程序,在 applet 中绘制一个金字塔。
- 9.21 编写一个程序,在 applet 中绘制一个正方体。
- 9.22 修改图 9.18 中的程序,在显示每种字体的名称时使用该字体,同时打印出每种字体的度量元。
- 9.23 在练习 1.21 中,我们编写过一个 applet,请用户输入一个圆的半径,再显示出圆的直径、周长和面积。修改练习 1.21 的程序,读入一系列坐标和半径,接着绘制该圆并显示该圆的半径、周长和面积。
- 9.24 编写一个 applet,模拟一个屏幕保护程序。在程序中随机地绘制线条,在绘制 100 条线后,自动清除屏幕并重新开始绘制。
- 9.25 修改练习 9.24 的程序,使程序在清除屏幕并重绘之前允许用户输入随机线条的数目。使用一个文本字段来获取该值,使得用户能够在程序执行的任意时刻向文本字段内键入新值。
- 9.26 修改练习 9.25 的程序,随机选择要显示的不同形状。
- 9.27 编写一个练习 4.38 (汉诺塔)的图形版本。在学习了第 14 章之后,读者就能使用 Java 的图像、动画和声音功能来实现这个练习的另一个版本。
- 9.28 修改练习 5.15 的掷骰子程序,在每次滚动后更新每个面的计数。使用 Graphics 类的 `drawString` 方法来输出总和。
- 9.29 修改练习 5.21 的程序(龟图),使用文本字段和按钮来丰富图形用户界面,利用绘制线条来代替绘制星号(\*)。当龟图程序指明了一次移动时,将位置数转换成屏幕上的相应像素数,这可以通过位置数乘以 10 (或其他想要的值)来实现。
- 9.30 制作一个骑士旅行问题(练习 5.22、练习 5.23、练习 5.26)的图形版本。在每次移动后,棋盘上相应的方格应根据合适的移动次数而进行更新。如果程序的结果为一个完整旅行

或一个封闭旅行，则程序应当显示相应的消息。

- 9.31 编写一个模拟龟兔赛跑（练习 5.41）的图形版本，绘制一段圆弧来模拟山峰，该圆弧从应用程序的左下角延伸到右上角，乌龟和兔子应当爬山。试着实现图形输出，将龟兔的每次移动打印在圆弧上。
- 9.32 制作一个走迷宫问题（练习 5.38 ~ 练习 5.40）的图形版本。使用已经生成的迷宫作为创建图形版本的指导。当走出该迷宫时，应当在迷宫中显示一个小圆，以指明当前的位置。
- 9.33 制作一个桶排序（练习 5.28）的图形版本，显示将每个值放入桶中以及最终复制回原始数组中的情形。

## 第10章 图形用户界面组件（一）

### 教学目标

- 理解图形用户界面（GUI）的设计原理
- 理解 Java.awt 软件包的类层次结构
- 学会构造图形用户界面
- 学会创建和使用按钮、标签、列表、文本字段、文本区域和面板等组件
- 理解并学会处理鼠标事件及键盘事件
- 理解并学会使用布局管理器

### 10.1 简介

图形用户界面（GUI）就是为应用程序提供一个图形化的界面。GUI能够使一个应用程序具有与众不同的“外观”和“感觉”。有了GUI，用户就不必花很多时间去记忆各个键盘序列的功能，从而把更多的时间投入在有效地使用应用程序本身上，用户和程序之间通过GUI进行交互。图10.1中给出了Netscape Navigator浏览器的GUI。



图 10.1 Netscape Navigator 浏览器的 GUI

GUI是由若干GUI组件（有时称为子组件）组成的。GUI组件是可见的对象，用户可以通过鼠标或键盘对它进行操作。图10.2中列出了几种GUI组件，在后面各节中，我们将详细讨论每一种GUI组件，下一章将介绍更高级的GUI组件。

用于创建GUI组件的类包含在Java.awt（abstract windowing toolkit）软件包中。可以通过下面的语句来引入java.awt软件包中的类：

```
import java.awt.* ;
```

其中星号(\*)是一个通配符,表示要引入的多个类。图 10.3 中列出了图 10.2 中各组件的继承层次结构,箭头方向表示派生关系。

| GUI 组件           | 说明                |
|------------------|-------------------|
| 标签 (Label)       | 用于显示文本 (该文本不可编辑)  |
| 按钮 (Button)      | 用鼠标在其上点击,则会激活一个事件 |
| 列表 (List)        | 用于显示一系列信息         |
| 文本字段 (TextField) | 用于从键盘输入文本或显示文本信息  |
| 面板 (Panel)       | 用于放置多个组件          |

图 10.2 基本的 GUI 组件

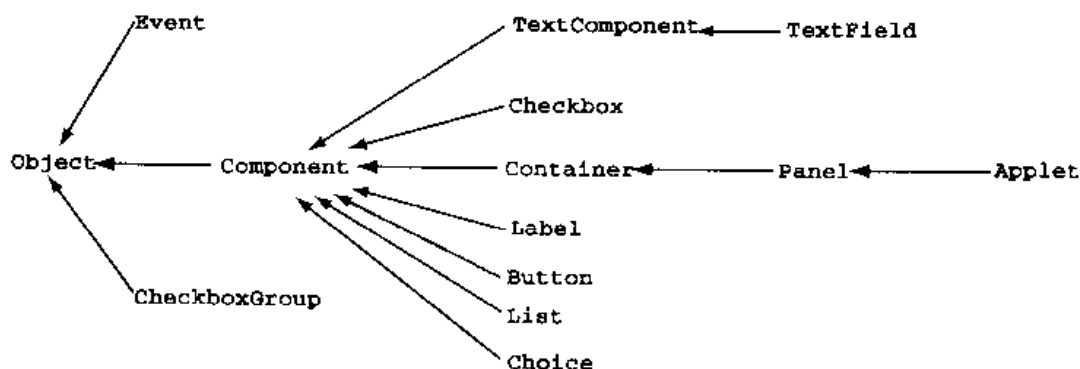


图 10.3 java.awt 软件包的部分继承层次结构

为了有效地使用 GUI 组件,必须了解 Java.awt 软件包的继承层次结构——尤其是 Component 类和 Container 类,GUI 组件的很多功能都是从这两个类派生而来的。任何从 Component 类继承的类都是组件。Label 类是从 Component 类继承的,而 Component 类又是从 Object 类继承的,所以 Label 既是组件又是对象,但 Component 仅仅是对象。同样,任何从 Container 类继承的类都是一个容器。容器是用于放置多个组件的一个区域。

#### 可移植性提示 10.1

一个 GUI 组件的外观可以因应用平台的不同而变化。随着应用平台的不同,Java 也具有不同的 GUI。

## 10.2 标签

在 GUI 上显示文本命令或信息的一种方式是使用标签。在标签上可以显示一行静态 (只读) 文本信息,这里的“静态”是指用户不能修改这些文本。标签使用 Label 类来创建,而 Label 类是从 Component 类直接派生的。如同其他对象一样,标签对象也必须通过调用构造函数来创建,图 10.4 中给出了两个构造函数。

| Label 类的构造函数                                              |
|-----------------------------------------------------------|
| public Label ()<br>创建一个“空”标签,即不显示任何文本                     |
| public Label (String s) //Label text<br>创建一个标签,标签上显示字符串 s |

图 10.4 Label 类的构造函数



图 10.5 中的程序创建了两个标签，并将它们加入到 applet 中，其中用到了图 10.4 的两个 Label 构造函数。

```
1 // Fig. 10.5: MyLabel.java
2 // Demonstrating the Label class constructors.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyLabel extends Applet {
7     private Font f;
8     private Label noLabel, textLabel;
9
10    public void init()
11    {
12        f = new Font( "Courier", Font.BOLD, 14 );
13
14        // call label constructor with no text
15        noLabel = new Label();
16
17        // call label constructor with a string argument
18        textLabel = new Label( "This is read-only text" );
19
20        // set font for text displayed in label
21        textLabel.setFont( f );
22
23        // add label components to applet container
24        add( noLabel );
25        add( textLabel );
26    }
27 }
```

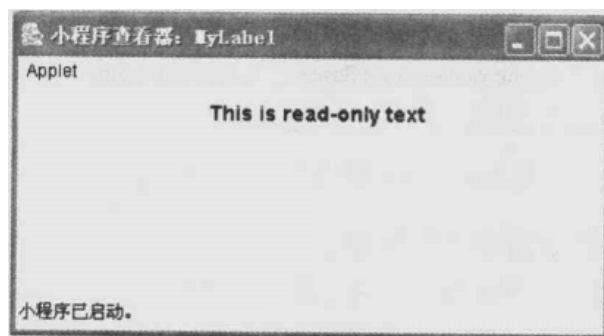


图 10.5 Label 类构造函数

Label 对象通过 Label 类及一个 Label 构造函数创建。下面的语句是使用 Label 类创建两个标签引用：

```
private Label noLabel, textLabel;
```

这两个标签对象都是在 init 方法中创建的。对象 noLabel 使用下面创建标签的语句进行创建：

```
noLabel = new Label ( );
```

这个标签上未显示任何文本。一般情况下，对于在初始时刻不必显示任何文本的标签通常使用这种类型的标签对象。不过，在后面的程序中，仍然可以在该标签上显示信息。在下面的例子（如

图 10.7 所示) 中, 我们将介绍如何为这种类型的标签设置文本。

标签对象 `textLabel` 通过下面的语句进行创建:

```
textLabel = new Label ( "This is read-only text");
```

在这个标签上显示文本字符串 “This is read-only text”。标签文本的字体由 `Component` 的 `setFont` 方法设置, 该方法使用一个 `Font` 对象作为参数。

要想使各标签对象可见, 必须使用 `add` 方法将它们添加到一个容器中。前面我们曾经介绍过, 容器是用于放置多个组件的一个区域。本章讨论的两种容器分别是 `applet` 和面板, 关于面板的详细内容请参看 10.8 节。在下一章, 我们会介绍更多的容器。使用 `Container` 的 `add` 方法可将组件安置到容器上, 下列语句:

```
add ( noLabel );
```

将标签 `noLabel` 添加到 `applet` 中。注意, `Label` 对象 `noLabel` 不包含任何文本。

#### 常见编程错误 10.1

忘记将组件添加到容器中会在运行时产生逻辑错误。

#### 常见编程错误 10.2

将还未创建的组件添加到容器中会抛出 `NullPointerException` 异常。

标签有几种相关的方法, 它们都列在图 10.6 中。`getText` 方法的功能是返回标签所显示的文本, 如果一个标签不包含任何文本, 那么就返回空字符串。`setText` 方法的功能是设置标签上的文本, 标签上原来显示的文本都替换成新设置的文本。

| Label 类的方法 <code>getText</code> 和 <code>setText</code>                                 |
|----------------------------------------------------------------------------------------|
| <code>public String getText ()</code><br>返回标签的文本                                       |
| <code>public void setText ( String s ) // read-only string to display</code><br>设置标签文本 |

图 10.6 Label 类的方法 `getText` 和 `setText`

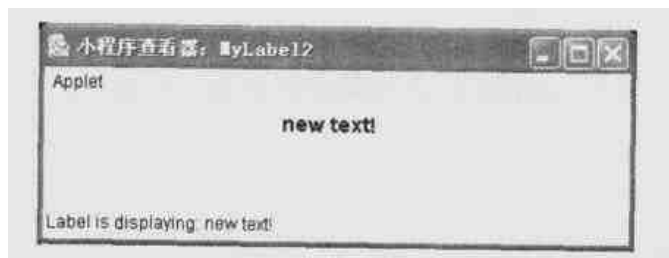
图 10.7 的程序说明了图 10.6 中两个方法的调用方式。在这个程序中, 首先创建了一个未显示任何文本的 `Label` 对象, 然后使用 `setText` 方法为该标签设置文本。`getText` 方法用于获取该标签的文本, 并显示在状态栏中。

```
1 // Fig. 10.7: MyLabel2.java
2 // Demonstrating Label class getText and setText methods.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyLabel2 extends Applet {
7     private Font f;
8     private Label noLabel;
9
10    public void init()
11    {
12        f = new Font( "Courier", Font.BOLD, 14 );
13
14        // call label constructor with no text
```

```

15         noLabel = new Label();
16
17         // set text in noLabel
18         noLabel.setText( "new text!" );
19
20         // set font for noLabel
21         noLabel.setFont( f );
22
23         // add label component
24         add( noLabel );
25     }
26
27     public void paint( Graphics g )
28     {
29         // get noLabel's text
30         showStatus( "Label is displaying: " + noLabel.getText() );
31     }
32 }

```

图 10.7 `getText` 和 `setText` 方法调用方式

开始时，对象 `noLabel` 上不包含任何文本。然后，将标签上要显示的文本字符串作为参数传送给 `setText` 方法。

下面这条语句的功能是在状态栏上显示 `noLabel` 的文本：

```
showStatus ( " Label is displaying : " + noLabel.getText ( ) );
```

`getText` 方法用于获取 `noLabel` 的文本。该语句放在 `Paint` 方法中，因此 `showStatus` 方法就可以在 applet 初始化之后显示 `noLabel` 的文本。如果这个语句是放在 `init` 方法中，那么由 `showStatus` 方法显示的信息就会由 “Applet started.” 所覆盖。

### 10.3 掀压式按钮

当用户通过鼠标点击按钮组件时，会激活某一个事件。按钮分为3种类型：掀压式按钮、选择按钮和复选框。在后面几节中，我们将讨论选择按钮和复选框。

当用鼠标点击掀压式按钮时，会激活一个事件。掀压式按钮由 `Button` 类创建，而 `Button` 类是由 `Component` 类直接派生的。图 10.8 列出了 `Button` 类的构造函数。

| Button 类的构造函数                           |                   |
|-----------------------------------------|-------------------|
| <code>public Button ( )</code>          | //no button label |
| 创建一个无标识的掀压式按钮                           |                   |
| <code>public Button ( String s )</code> | //button label    |
| 创建一个有标识的掀压式按钮                           |                   |

图 10.8 `Button` 类的构造函数

按钮表面上显示的文字称为按钮标签。一个GUI上可以包含很多按钮,但每个按钮的标签必须是惟一的,这是因为按钮是事件驱动的。当点击按钮时,就会产生一个按钮事件。按钮标签通常用于确定所按下的是哪一个按钮,如果两个按钮的标签相同,就会使用户产生混淆。

#### 常见编程错误 10.3

多个按钮具有相同的标签是一个逻辑错误,这样的按钮会使用户产生混淆。

#### 编程技巧 10.1

为每个按钮提供一个惟一的名称。

在上一节中,我们没有讨论事件处理,因为通过鼠标点击一个标签不会产生事件。GUI是由事件驱动的,也就是它们根据事件来执行相应的动作。事件就是动作,比如移动鼠标、点击鼠标等。事件由窗口系统发送到Java程序,存放在Event类的对象中。Event类是由Object类直接派生的,这些事件由Component类的方法进行处理。Event类也是java.awt软件包的一部分。在图10.2的组件中,除了标签和面板之外,其他组件都能产生事件,我们将在介绍每一种组件的同时讨论它们相应的事件处理方法。

图10.9的程序创建了3个按钮,并将它们添加到applet中。其中的两个按钮带有按钮标签。在这个applet的action方法中,为每个按钮都提供了相应的事件处理程序,当按下某个按钮时,就会完成某一个动作。下面的语句用于创建3个Button引用:

```
private Button pushButton1, pushButton2, pushButton3;
```

每个Button引用都通过new和一个Button构造函数来实例化。下面的语句分别对pushButton1和pushButton2进行实例化:

```
pushButton1 = new Button ( " Click here " );  
pushButton2 = new Button ( "Sorry I don't do anything" );
```

构造函数Button中的字符串参数就是按钮标签,其中对象pushButton3没有标签,这些按钮通过Container类的add方法连接到applet中。下面的语句可将3个按钮设置到applet中:

```
add ( pushButton 1 );  
add ( pushButton 2 );  
add ( pushButton 3 );
```

使用鼠标点击任何一个按钮都会产生一个事件。

```
1 // Fig. 10.9: MyButtons.java  
2 // Creating push buttons.  
3 import java.applet.Applet;  
4 import java.awt.*;  
5  
6 public class MyButtons extends Applet {  
7     private Button pushButton1, pushButton2, pushButton3;  
8  
9     public void init()  
10    {  
11        pushButton1 = new Button( "Click here" );  
12        pushButton2 = new Button( "Sorry I don't do anything" );  
13        pushButton3 = new Button(); // no button label  
14    }
```

```

15         // add buttons
16         add( pushButton1 );
17         add( pushButton2 );
18         add( pushButton3 );
19     }
20
21     // handle the button events
22     public boolean action( Event e, Object o )
23     {
24         // check to see if a button triggered the event
25         if ( e.target instanceof Button ) {
26
27             // check to see if pushButton1 or pushButton3
28             // was pressed. Nothing will be done if
29             // pushButton2 was pressed.
30             if ( e.target == pushButton1 )
31                 showStatus( "You pressed: " + o.toString() );
32             else if ( e.target == pushButton3 )
33                 showStatus( "You pressed: " + e.arg );
34
35             return true;    // event was handled here
36         }
37
38         return true;
39     }
40 }

```



图 10.9 创建3个按钮

当点击按钮时,就会调用 `action` 方法。`action` 方法返回一个布尔值,并带有两个参数——`Event` 和 `Object`。`Event` 保存了关于事件及产生该事件的组件的信息,参数 `Object` 保存了组件的相关信息(对于按钮来说,表示按钮标签)。为使按钮能完成指定的任务,必须使用 `MyButtons` 类替换 `Component` 类的 `action` 方法。

在 `action` 方法中,第一条 `if` 语句的条件:

```
e.target instanceof Button
```

用于判断产生事件的是否是按钮。Event 类的实例变量 target 表示产生事件的组件。运算符 instanceof 用于判断 e.target 是否是一个 Button 对象。如果是，则返回 true 值。由于该程序中包含多个按钮，因此我们需要知道按下的是哪个按钮。下一条 if 语句的条件：

```
e.target == pushButton1
```

用于判断产生事件的是否是 pushButton1 按钮。下面的语句将显示字符串 “You pressed:Click here”：

```
showStatus ( " You pressed: " + o.toString ( ) );
```

对象 o 中保存的是按钮标签，除此之外没有其他信息。最后一条 if 语句的条件：

```
e.target == pushButton3
```

用于判断产生事件的是否是 pushButton3。下面的语句将显示字符串 “You pressed:”：

```
showStatus( "You pressed:" + e.arg );
```

Event 类的实例变量 arg（数据类型为 Object）包含按钮标签。值 e.arg 与 o 总是是一致的。在后面的例子中将使用 e.arg。

最后一个 if 条件也可以使用 e.arg 重写：

```
e.arg.equals ( pushButton3.getLabel ( ) )
```

Button 类的方法 getLabel 返回一个按钮标签作为 String 对象。Object 类的方法 equals 用于比较 arg 和 pushButton3 的按钮标签。

这个程序的 action 方法仅用于处理 Button 事件。具体地说，action 方法处理的是 pushButton1 和 pushButton3 产生的事件。当点击按钮 pushButton2 时，它也会产生一个事件，但程序中没有对该事件进行处理。如果一个按钮产生了事件，那么返回 true 值就表示 action 方法中已经处理了该事件。在下一章中，我们将更详细地讨论事件处理。

## 10.4 文本字段

文本字段是一个单行显示区域，可以从键盘上接收用户输入。用户将数据输入文本字段，然后点击回车键就可以在程序中使用该数据。通过 TextField 类创建的文本字段也可用于显示信息，TextField 类是由 TextComponent 类直接派生的。图 10.10 中列出了 TextField 类的构造函数。

| TextField 类的构造函数                            |                                                                                            |
|---------------------------------------------|--------------------------------------------------------------------------------------------|
| public TextField ( )                        | 创建一个 TextField 对象                                                                          |
| public TextField ( int columns )            | //number of Columns<br>创建一个空的 TextField 对象，并指定列宽                                           |
| public TextField ( String s )               | //text displayed in text field<br>创建一个 TextField 对象，显示字符串 s                                |
| public TextField ( String s , int columns ) | // text displayed in text field<br>// number of columns<br>创建一个 TextField 对象，以指定的列宽显示字符串 s |

图 10.10 TextField 类的构造函数

## 常见编程错误 10.4

如果将 `TextField` 的 “F” 写成小写的 “f”，则会产生编译错误。

在图 10.11 的程序中，使用 `TextField` 类和 `TextField` 构造函数创建了 4 个文本字段。这个程序没有使用 `action` 方法来处理文本字段事件，我们将在下一个例子中说明如何处理该问题。在这 4 个文本字段中输入信息不会产生任何作用。

```
1 // Fig. 10.11: MyTextfield.java
2 // Demonstrating the TextField class constructors.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyTextfield extends Applet {
7     private TextField text1, text2, text3, text4;
8
9     public void init()
10    {
11        // construct textfield with default sizing
12        text1 = new TextField();
13
14        // construct textfield with default text
15        text2 = new TextField( "Some text " );
16
17        // construct textfield with 25 elements visible
18        text3 = new TextField( 25 );
19
20        // construct textfield with default text and
21        // 40 visible elements
22        text4 = new TextField( "Yet some more text", 40 );
23
24        // add textfields to applet
25        add( text1 );
26        add( text2 );
27        add( text3 );
28        add( text4 );
29    }
30 }
```

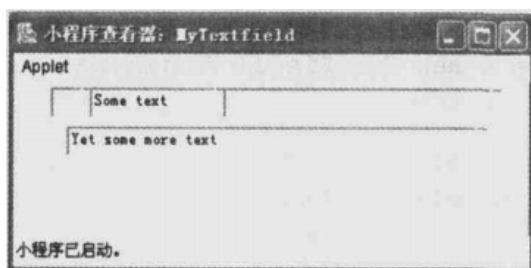


图 10.11 `TextField` 类的构造函数

通过下列语句：

```
text1=new TextField ( );
```

创建 `TextField` 的对象 `text1`。这个文本字段没有显示任何文本，默认的列宽为 2，列宽是指字符的平

均宽度。尽管 text1 只有两列宽，但用户仍可以输入两个以上的字符，不过在显示时一次只能看见两个字符。无论文本字段的列宽是多少，都可以使用方向键在文本字段中来回滚动，以浏览文本字段中的所有信息。通过下列语句：

```
text2=new TextField ( " Some text " ) ;
```

创建 TextField 的对象 text2。该文本字段的列宽为所显示字符串的长度。通过下列语句：

```
text3=new TextField ( 25 );
```

创建 TextField 对象 text3。这个文本字段的列宽为 25，未显示任何文本。TextField 对象 text4 的列宽为 40，并显示字符串 "Yet some more text"。这些文本字段都是通过 Component 的方法 add 添加到 applet 中的。

图 10.12 的程序包含 3 个文本字段。其中两个用于显示文本，另一个用于接收用户的输入。在接收输入的文本字段中，所输入的字符将隐藏起来。

这 3 个 TextField 对象是通过 TextField 构造函数进行创建和初始化的。下面使用方法 setEditable 将文本字段 message 和 message2 设为只读：

```
message.setEditable ( false );
message2.setEditable ( false );
```

文本字段 passwordField 所用的掩码字符是由 TextField 的方法 setEchoCharacter 设置的。每当在文本字段 passwordField 中输入一个字符时，就会显示一个星号 (\*)。在文本字段中按下回车键会激活一个事件。

---

```

1 // Fig. 10.12: MyTextField2.java
2 // Demonstrating TextField events.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyTextField2 extends Applet {
7     private TextField message, passwordField, message2;
8     private String password;
9
10    public void init()
11    {
12        password = "Marak954"; // set password
13        message = new TextField( "Enter password: " );
14        message.setEditable( false ); // read only
15
16        passwordField = new TextField( 12 );
17        passwordField.setEchoCharacter( '*' ); // masking char
18
19        message2 = new TextField( 30 );
20        message2.setEditable( false );
21
22        // add textfields to applet
23        add( message );
24        add( passwordField );
25        add( message2 );
26    }
27
```



```

28     public boolean action( Event e, Object o )
29     {
30         // check to see if a textfield triggered the event
31         if ( e.target instanceof TextField )
32
33             // check to see it was the textfield passwordField
34             if ( e.target == passwordField )
35
36                 // check for correct password
37                 if ( e.arg.equals( password ) )
38                     message2.setText( "Access Granted" );
39                 else
40                     message2.setText( "Invalid password. " +
41                                     "Access Denied" );
42
43         return true;    // event was handled here
44     }
45 }

```



图 10.12 TextField 事件

在这个 applet 中, 程序重写了 action 方法, 以处理由文本字段产生的事件。任何文本字段都会产生文本字段事件——即使是只读文本字段也会产生这种事件。尽管 message 和 message2 都是只读的, 但用户仍可以在这两个文本字段中点击回车键, 从而产生事件。

当产生一个文本字段事件时, Event 的实例变量 arg 和 Object (action 方法的输入参数) 各包含一个字符串, 它代表生成该事件的文本字段的文本。

下而的判断条件用于判断产生事件的文本字段是否是 passwordField:

```
e.target == passwordField
```

如果产生事件的文本字段是 passwordField, 那么就要进行下而的判断:

```
e.arg.equals( password )
```

这个条件用于判断密码是否正确, 实例变量 arg 的值是用户在文本字段中输入的文本。如果密码正确, 那么在文本字段 message2 中就会显示字符串 "Access Granted"。否则, message2 中就会显示 "Invalid Password. Access Denied"。

## 10.5 选择按钮

选择按钮提供了一系列选项，用户可以从中进行选择。选择按钮由 Choice 类创建，Choice 类是从 Component 类直接派生的。Choice 类的构造函数未被重载，而且没有参数。图 10.13 列出了 Choice 类的一些通用方法。

| Choice 类的方法                                                                  |
|------------------------------------------------------------------------------|
| <code>public int countItems ()</code><br>返回选择按钮中的选项个数                        |
| <code>public String getItem ( int index )</code><br>返回带有指定下标的 Choice 选项      |
| <code>public Synchronized void addItem ( String s )</code><br>向选择按钮中添加一个选项 s |
| <code>public String getSelectedItem ()</code><br>返回选中的 Choice 选项             |
| <code>public int getSelectedIndex ()</code><br>返回选中的 Choice 选项的下标            |

图 10.13 Choice 类的方法

图 10.14 的程序通过一个选择按钮提供了 3 个字体选项。当选中一个字体时，文本字段中的文本就变为该字体。这个程序使用了图 10.13 中的方法。

下面的语句使用 Choice 构造函数来创建一个选择按钮对象：

```
choiceButton = new Choice ( ) ;
```

程序中使用 addItem 方法来添加选择按钮中的选项，即将一个字符串添加到选择按钮中。选择按钮中的选项根据添加的顺序进行排列，并使用一个数字类型的下标来表示它们的顺序。第一个添加的选项下标为 0，第二个添加的选项的下标为 1，依次类推。当把 Choice 按钮添加到一个容器中时，就会显示出该 Choice 按钮的第一个选项，点击向下箭头可以选择其他选项。当点击向下箭头时，会弹出一个选项列表供用户进行选择。

```
1 // Fig. 10.14: MyChoice.java
2 // Using a Choice button to select a font.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyChoice extends Applet {
7     private Choice choiceButton;
8     private TextField t;
9     private Font f;
10
11     public void init()
12     {
13         choiceButton = new Choice();
14         t = new TextField( "Sample Text", 16 );
15         t.setEditable( false );
16
17         // add items to choiceButton
18         choiceButton.addItem( "TimesRoman" );
19         choiceButton.addItem( "Courier" );
```

```

20         choiceButton.addItem( "Helvetica" );
21
22         f = new Font( choiceButton.getItem( 0 ),
23                       Font.PLAIN, 14 );
24         t.setFont( f );
25         add( choiceButton );
26         add( t );
27     }
28
29     public boolean action( Event e, Object o )
30     {
31         String s;
32
33         // Check for Choice button event
34         if ( e.target instanceof Choice ) {
35             f = new Font( choiceButton.getSelectedItem(),
36                           Font.PLAIN, 14 );
37
38             t.setFont( f );
39
40             s = "Number of items: " +
41               choiceButton.countItems();
42
43             s += "      Current index: " +
44               choiceButton.getSelectedIndex();
45
46             showStatus( s );
47         }
48
49         return true;
50     }
51 }

```

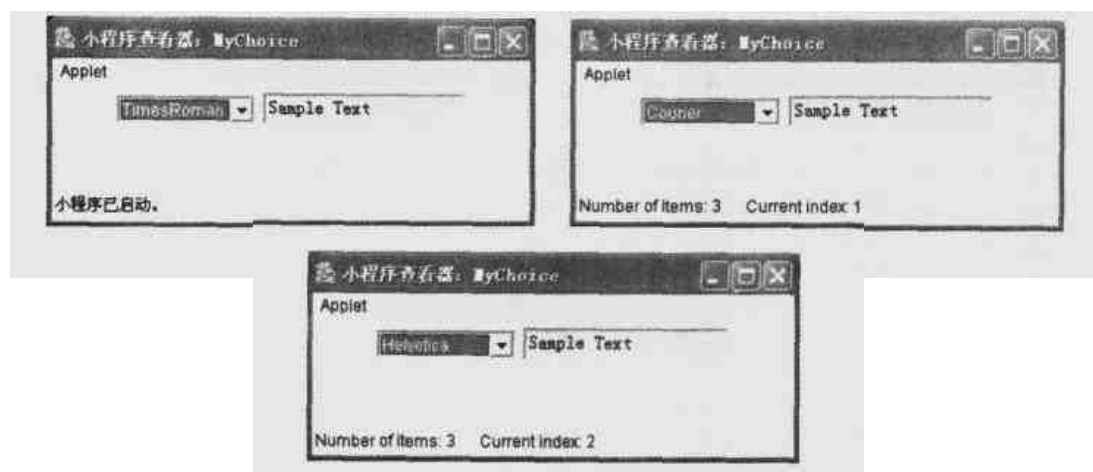


图 10.14 通过选择按钮来选择字体

通过下列语句:

```
f = new Font ( choiceButton.getItem ( 0 ), Font.PLAIN, 14 );
```

创建了 Font 对象 f。getItem 方法用于返回下标为 0 的选项——“Times Roman”。可以使用 setFont 方法来设置文本字段中的字体。

在这个 applet 中，程序重写了 action 方法，这样在选择按钮中选择选项时会产生相应的事件。通过下列语句：

```
f = new Font ( choiceButton.getSelectedItem ( ) , Font.PLAIN , 14 );
```

创建了 Font 对象 f，getSelectedItem 方法返回的是选中的选项，用于变换字体。上面的语句也可以写成下面的形式：

```
f = new Font ( e.arg , Font.PLAIN, 14 );
```

Event 实例 arg 的值是一个字符串，表示所选中的选项，使用 countItems 方法计算选择按钮中选项的个数，并显示在状态栏中；使用 getSelectedIndex 方法计算当前选中的选项的下标，同样也显示在状态栏中。

## 10.6 复选框按钮和单选按钮

可以通过 Checkbox 类创建复选框按钮和单选按钮，它们都是状态按钮（也就是这些按钮具有 on/off 或 true/false 值）。我们首先讨论复选框按钮。

Checkbox 类是从 Component 类直接派生的。图 10.15 列出了 Checkbox 类的构造函数。

| Checkbox 类的构造函数                                                 |
|-----------------------------------------------------------------|
| public Checkbox ()<br>创建一个无标签的复选框对象，该复选框的初始状态为非选中状态             |
| public Checkbox (String s)<br>创建一个带有标签 s 的复选框对象，该复选框的初始状态为非选中状态 |

图 10.15 Checkbox 类的构造函数

图 10.16 的程序使用复选框对象来变换文本字段中所显示文本的字体风格。其中一个复选框表示黑字体，另一个复选框表示斜字体。当该程序开始运行时，两个复选框都为非选中状态。在 HTML 文件中，将 applet 的大小设置为 290 × 70。

```

1 // Fig. 10.16: MyCheckbox.java
2 // Creating Checkbox buttons.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyCheckbox extends Applet {
7     private Font f;
8     private TextField t;
9     private Checkbox checkBold, checkItalic;
10
11     public void init()
12     {
13         t = new TextField( "Sample Text", 30 );
14
15         // instantiate checkbox objects
16         checkBold = new Checkbox( "Bold" );
17         checkItalic = new Checkbox();
18         checkItalic.setLabel( "Italic" ); // set checkbox label

```

```

19
20     f = new Font( "TimesRoman", Font.PLAIN, 14 );
21     t.setFont( f );
22
23     add( t );
24     add( checkBold );    // unchecked (false) by default
25     add( checkItalic );  // unchecked (false) by default
26 }
27
28 public boolean action( Event e, Object o )
29 {
30     int b, i;
31
32     // Check for Checkbox event
33     if ( e.target instanceof Checkbox ) {
34
35         // test state of bold checkbox
36         if ( checkBold.getState() == true )
37             b = Font.BOLD;
38         else
39             b = Font.PLAIN;    // value of 0
40
41         // test state of italic checkbox
42         if ( checkItalic.getState() == true )
43             i = Font.ITALIC;
44         else
45             i = Font.PLAIN;    // value of 0
46
47         f = new Font( "TimesRoman", b + i, 14 );
48         t.setFont( f );
49     }
50
51     return true;
52 }
53 }

```

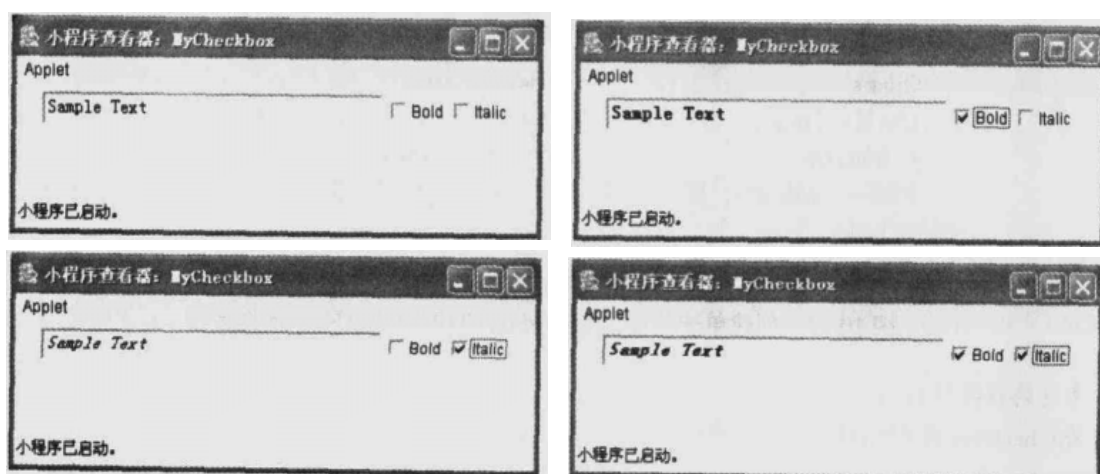


图 10.16 创建一组 (2个) 复选框按钮

通过下列语句:

```
private Checkbox checkBold ,checkItalic;
```

声明了 Checkbox 引用。通过下列语句：

```
checkBold=new Checkbox ( " Bold " );
```

创建对象 checkBold。构造函数所带有的参数 Bold 是复选框标签，复选框标签显示在复选框的右侧。下面的语句用于创建对象 checkItalic，并为其设置标签：

```
checkItalic=new Checkbox ( ) ;
checkItalic.setLabel ( " Italic " );
```

在创建 checkItalic 时，它并不带有标签，checkItalic 对象标签是由 Checkbox 类的 setLabel 方法设置的。通过这种方式创建 checkItalic 的目的是为了说明无参构造函数及 setLabel 的使用方法。另外，通过 add 方法将文本字段和各个复选框添加到 applet 中。

在这个 applet 中，程序重写了 action 方法。因此，当使用鼠标点击 Checkbox 对象时，会激活相应的事件。Event 类的实例变量 target 用于判断产生事件的组件是否为复选框，复选框的值为 true（此时复选框中带有选中标签）或 false（此时复选框为空）。每个复选框对象的状态由后面的 if/else 语句判断，这里必须使用两个 if/else 语句，因为这两个复选框彼此是独立的，选中一个不会影响另一个。因此，一次可以同时选中多个复选框。下面的语句用于判断复选框 checkItalic 的状态：

```
checkItalic.getState ( ) == true
```

Checkbox 类的 getState 方法返回一个布尔值，表示复选框的状态。在这里也可以使用 Event 类的实例 arg；对于复选框来说，其 arg 是一个布尔对象，必须用 Boolean 方法 booleanValue 将其转换成布尔值。一旦选中了某个复选框，文本字段中的文字就将更新为相应的字体。

还可以将复选框组合到一起，成为一组单选按钮，在这组按钮中，每次只能选中其中的一个，其他复选框的值都为 false（即未选中）。单选按钮由 CheckboxGroup 类和 Checkbox 类创建。CheckboxGroup 类是从 Object 类直接派生的，而不是从 Component 类派生的。这意味着 CheckboxGroup 对象不能添加到容器中，因为只有从 Component 类派生的类对象才能添加到容器中。图 10.17 中列出了用于创建单选按钮的构造函数。

| 创建单选按钮的 Checkbox 和 CheckboxGroup 构造函数                 |                                     |
|-------------------------------------------------------|-------------------------------------|
| public Checkbox (                                     |                                     |
| String s                                              | //radio button label                |
| CheckboxGroup c,                                      | //CheckboxGroup owning radio button |
| boolean state )                                       | //state of radio button             |
| 创建一个单选按钮，其标签为 s，状态为 state。该单选按钮将添加到 CheckboxGroup c 中 |                                     |
| public CheckboxGroup ( )                              |                                     |
| 创建一个 CheckboxGroup 对象                                 |                                     |

图 10.17 创建单选按钮的 Checkbox 和 CheckboxGroup 构造函数

#### 常见编程错误 10.5

将 Checkbox 或 CheckboxGroup 中的“b”写成大写“B”会引发语法错误。

#### 常见编程错误 10.6

向容器中添加 CheckboxGroup 对象会产生编译错误。

图 10.18 的程序和前面的程序类似。用户可以修改文本字段中的文本字体。在这个程序中使用的是单选按钮，即只允许进行单项选择。

```
1 // Fig. 10.18: RadioButton.java
2 // Creating radio buttons using CheckboxGroup and Checkbox.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class RadioButton extends Applet {
7     private TextField t;
8     private Font f;
9     private CheckboxGroup radio;
10    private Checkbox radioBold, radioItalic,
11           radioPlain;
12
13    public void init()
14    {
15        t = new TextField( "Sample Text", 40 );
16
17        // instantiate checkbox group (i.e. radio buttons)
18        radio = new CheckboxGroup();
19
20        add( t ); // add textfield
21
22        // instantiate radio button objects
23        add( radioPlain = new Checkbox( "Plain", radio, true ) );
24        add( radioItalic = new Checkbox( "Italic", radio, false ) );
25        add( radioBold = new Checkbox( "Bold", radio, false ) );
26    }
27
28
29
30    public boolean action( Event e, Object o )
31    {
32        int style;
33
34        // Check for Checkbox event
35        if ( e.target instanceof Checkbox ) {
36
37            // test state of radio buttons
38            if ( radioPlain.getState() == true )
39                style = Font.PLAIN;
40            else if ( radioItalic.getState() == true )
41                style = Font.ITALIC;
42            else
43                style = Font.BOLD;
44
45            f = new Font( "TimesRoman", style, 14 );
46            t.setFont( f );
47        }
48
49        return true;
50    }
51 }
```

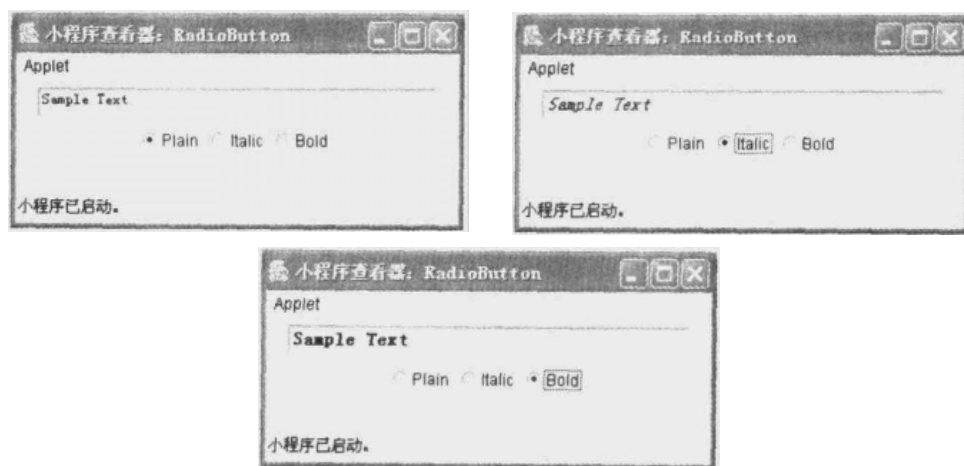


图 10.18 创建单选按钮

下面的语句用于创建一个名为 `radio` 的 `CheckboxGroup` 引用：

```
private CheckboxGroup radio;
```

本例中另外还创建了三个 `Checkbox` 引用。下面的语句用于对 `radio` 进行实例化：

```
radio = new CheckboxGroup ( ) ;
```

由于 `CheckboxGroup` 类不是从 `Component` 类派生的，所以不能将对象 `radio` 添加到 applet 中，只有 `Component` 类的对象才能添加到 `Container` 类的对象中。所以，对象 `radio` 只是作为一个“纽带”将三个复选框连接在一起成为三个单选按钮。每个复选框都是一个组件，因此可以将它们添加到 applet 中。下面的三条语句用于将三个单选按钮分别添加到 applet 中：

```
add ( radioPlain=newCheckbox ( " Plain ", radio, true ) );
add ( radioItalic=newCheckbox ( " Italic ", radio, false ) );
add ( radioBold=newCheckbox ( " Bold ", radio, false ) );
```

现在，这些复选框就成为属于 `CheckboxGroup` 对象 `radio` 的单选按钮了。构造函数的第一个参数是单选按钮的标签，第二参数是拥有该单选按钮的 `CheckboxGroup`。最后一个参数表示单选按钮的初始状态——在一组单选按钮中，只能有一个设置为 `true`；但是，允许将所有单选按钮的初始值都设为 `false`。如果在同一个 `CheckboxGroup` 中将多个单选按钮设为 `true` 值，那么只有最后一个设置为 `true` 的单选按钮是有效的（选中的）。

#### 常见编程错误 10.7

当创建单选按钮时，如果将多个单选按钮的值设置为 `true`，则会产生运行时的逻辑错误。

在这个例子中仍然使用了 `action` 方法。`Event` 类的实例 `target` 用于判断是否为复选框事件，因为单选按钮也是一种复选框，`Checkbox` 类的 `getState` 方法用于获取单选按钮的状态。`Event` 类的实例 `arg` 在 `action` 方法中已经使用过，当一个单选按钮产生事件时，`arg` 实例变量就包含一个 `Boolean` 对象。确定单选按钮的状态后，就将文本字段中的文本设置为相应的字体。

## 10.7 列表

列表用于显示一系列的选项，用户可以从选择一个或多个选项。列表是通过 `List` 类创建的，



而 List 类则是从 Component 类直接派生的。图 10.19 中列出了 List 类的构造函数。

| List 类的构造函数                                       |                           |
|---------------------------------------------------|---------------------------|
| public List ()                                    | 创建一个列表对象, 该列表中只允许选中一个选项   |
| public List (                                     |                           |
| int items ,                                       | //number of items visible |
| boolean ms )                                      | //multiple selections     |
| 创建一个列表对象, 选项可见行数为 items, 如果 ms 为 true, 表示可以选择多个选项 |                           |

图 10.19 List 类的构造函数

图 10.20 中的程序创建了一个列表, 其中带有 4 个颜色选项。当使用鼠标双击列表中的一个颜色名称时, applet 的背景就转换为这种颜色。下列语句:

```
private List colorList;
```

用于创建引用 colorList。

```

1    // Fig. 10.20: MyList.java
2    // Creating a List.
3    import java.applet.Applet;
4    import java.awt.*;
5
6    public class MyList extends Applet {
7        private List colorList;
8
9        public void init()
10       {
11            // create a list with 5 items visible
12            // do not allow multiple selections
13            colorList = new List( 5, false );
14
15            // add four items to the list
16            colorList.addItem( "White" );
17            colorList.addItem( "Black" );
18            colorList.addItem( "Yellow" );
19            colorList.addItem( "Green" );
20
21            // add list to applet
22            add( colorList );
23        }
24
25        public boolean action( Event e, Object o )
26        {
27
28            // test for a list
29            if ( e.target instanceof List ) {
30
31                if ( e.arg.equals( "Yellow" ) )
32                    setBackground( Color.yellow );
33                else if ( e.arg.equals( "White" ) )
34                    setBackground( Color.white );
35                else if ( e.arg.equals( "Black" ) )
```

```
36         setBackground( Color.black );
37     else
38         setBackground( Color.green );
39
40     repaint();    // update applet
41 }
42
43     return true;
44 }
45 }
```

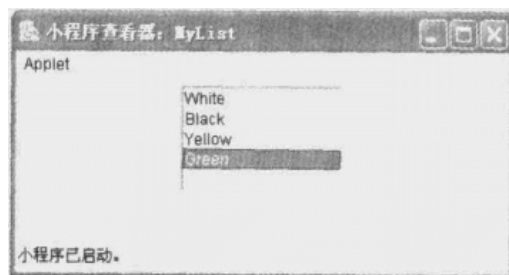


图 10.20 创建一个列表

通过下面的语句可以创建 colorList 对象：

```
colorList=new List ( 5, false );
```

该语句表示列表 colorList 的长度为 5 行，并且不允许选择多个选项。可以通过 addItem 方法添加列表中的选项，选项只能是字符串。列表是使用 add 方法添加到 applet 中的。

#### 常见编程错误 10.8

向列表中添加一个非字符串的选项会引发语法错误。

使用鼠标双击列表选项会产生一个事件。在这个例子中仍然使用了 action 方法。Event 的实例变量 target 用于判断产生事件的是否为列表，列表的 arg 实例变量表示列表中被选中的字符串。程序中使用 arg 实例判断列表中的每个字符串，并用 setBackground 方法来修改 applet 的背景颜色。

我们在第 9 章讨论了 paint 方法，并简要提到了 repaint 方法。repaint 方法调用了 Component 的 update 方法。update 方法利用当前的背景色来填充 applet 的背景，并调用 paint 方法，从而重画或刷新 applet。repaint 方法很有用，因为它自动提供了 update 和 paint 都需要使用的 Graphics 对象。

图 10.21 的程序声明了一个可多选列表，从该列表中可以将选项复制到另一个列表中。使用鼠标选中要复制的各个选项，然后点击两个列表对象中间的 Copy 按钮，这样就可以一次复制多个选项。注意，可以从列表的任意位置选择要复制的对象。

可以使用下面的语句创建对象 stateList：

```
stateList=new List ( 5, true );
```

构造函数的第一个参数表示列表的长度为 5 行，第二个参数表示该列表允许选择多个选项。列表 copyList 的长度也为 5 行，但它不允许多选。在两个列表之间还创建了一个按钮 Copy，该按钮用于复制列表中的选项。

在 stateList 对象中，通过 addItem 方法添加了 8 个选项。因为该列表一次只能使 5 个选项可见，因此在列表的右边自动出现了一个滚动条。点击滚动条两端的上、下方向键按钮，可以使列表中的内容上下滚动，以浏览整个列表的内容。仅当列表中的选项个数超过列表的可见长度时才会出现滚动条。

```
1 // Fig. 10.21: MyList2.java
2 // Copying items from one List to another.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyList2 extends Applet {
7     private List stateList, copyList;
8     private Button copy;
9
10    public void init()
11    {
12        // create a list with 5 items visible
13        // allow multiple selections
14        stateList = new List( 5, true );
15
16        // create a list with 5 items visible
17        // do not allow multiple selections
18        copyList = new List( 5, false );
19
20        // create copy button
21        copy = new Button( "Copy >>>" );
22
23        // add some data to stateList
24        stateList.addItem( "Texas" );
25        stateList.addItem( "Alaska" );
26        stateList.addItem( "Florida" );
27        stateList.addItem( "Montana" );
28        stateList.addItem( "Mississippi" );
29        stateList.addItem( "Delaware" );
30        stateList.addItem( "New Mexico" );
31        stateList.addItem( "South Dakota" );
32        stateList.addItem( "West Virginia" );
33
34        // add list to applet
35        add( stateList );
36        add( copy );
37        add( copyList );
38    }
39
40    public boolean action( Event e, Object o )
41    {
42        String states [ ];
43
44        // test for a button event
45        if ( e.target == copy ) {
46
47            // get the selected states
48            states = stateList.getSelectedItems();
49
50            // copy them to copyList
51            for ( int i = 0; i < states.length; i++ )
52                copyList.addItem( states[ i ] );
53        }
54
55        return true;
56    }
57 }
```

```

56     }
57 }

```

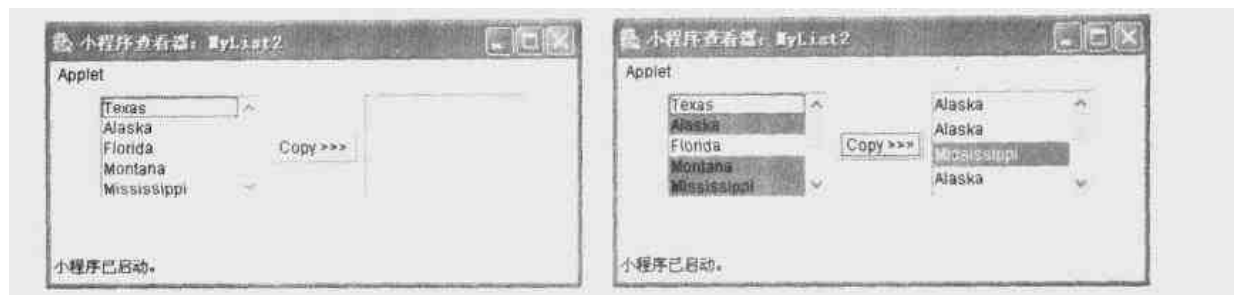


图 10.21 使用一个可多选的列表

在这个 applet 中，仍然使用 action 方法来处理事件。为了从一个可多选的列表选取选项，必须激活一个外部事件（也就是应该使用另一个组件来指明列表中进行了多项选择）。在本例中，使用一个按钮来激活该事件。Event 类的实例变量 target 用于判断产生事件的是否为 Copy 按钮，并在程序中处理该事件。注意，在一个可多选的列表中，双击一个选项也可能产生一个事件。

数组 states 中保存着选中的各列表选项的内容。下面的语句使用了 getSelectedItems 方法来获得列表中被选中的选项的内容：

```
states=stateList.getSelectedItems ( );
```

程序中使用了一个 for 循环，将数组 states 中的每一个元素都添加到 copyList 中。

## 10.8 面板

我们在前面曾经提到过，面板是一种容器。Panel 类是从 Container 类继承下来的，而 Applet 类又是从 Panel 类继承下来的。因此，Panel 和 Applet 都是容器，都可以包含其他的组件（包括其他的面板）。Panel 类的构造函数没有参数。

图 10.22 中的程序创建了两个面板，并将它们添加到 applet 中。通过修改面板的背景色可以将它们区分开来。在程序中，我们使用 Component 的方法 setBackground 将一个面板改成粉红色，而将另一个改成黄色。

```

1  // Fig. 10.22: TwoPanels.java
2  // Creating Panel objects.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class TwoPanels extends Applet {
7      private Panel p1, p2;
8      private Label label1, label2;
9
10     public void init()
11     {
12         p1 = new Panel();    // create a panel object
13         p2 = new Panel();    // create a second object
14         label1 = new Label( " first panel" );
15         label2 = new Label( "second panel" );
16

```

```

17         // change p1's background to yellow
18         p1.setBackground( Color.yellow );
19
20         // change p2's background to pink
21         p2.setBackground( Color.pink );
22
23         p1.add( label1 );    // add label to p1
24         p2.add( label2 );    // add label to p2
25
26         add( p1 );          // add panel to applet
27         add( p2 );          // add panel to applet
28     }
29 }

```

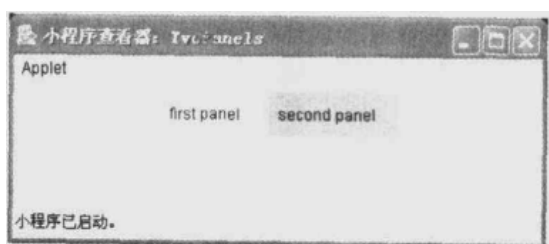


图 10.22 创建两个 Panel 对象

程序中创建了两个 panel 对象 p1 和 p2, 而且还创建了两个 Label 对象。这里使用了 add 方法分别将两个对象 Label 添加到两个面板上, 下面的语句是将 Label1 添加到面板 p1 上:

```
p1.add ( Label1 );
```

然后, 再将每个面板对象添加到 applet 上, 下面是将 p1 添加到 applet 上:

```
add ( p1 );
```

注意, 图 10.22 中面板的大小与它所包含的标签组件的大小一致, 不过面板的大小不会影响 applet 的大小, 通过使用多个面板, applet 可以创建复杂的 GUI。10.11 节中要介绍的布局管理器将会把面板上的组件安排得更精确。面板不会产生事件, 所以不必对它进行事件处理。

#### 常见编程错误 10.9

向一个还未实例化的容器中添加组件会产生 NullPointerException 异常。

## 10.9 鼠标事件

在这一节中将介绍一些用于处理鼠标事件的 Component 类的方法和常量。当用户通过鼠标进行交互操作时, 就会产生鼠标事件。图 10.23 中列出了所有的方法, 图 10.28 中列出了所有的常量, 我们将分别讨论每一个方法和常量。

#### Component 类的鼠标方法

```
public boolean mouseDown (
```

```
    Event e,
```

```
        //Event object
```

```
    int x,
```

```
        //x coordinate of mouse
```

```
    int y)
```

```
        //y coordinate of mouse
```

```
    处理“鼠标按下”事件, 当用户按下某个鼠标按键时会产生“鼠标按下”事件
```

(续表)

## Component 类的鼠标方法

```

public boolean mouseUp (
    Event e,                //Event object
    int x,                  //x coordinate of mouse
    int y,                  //y coordinate of mouse
    处理“鼠标释放”事件，当用户释放某个鼠标按键时会产生“鼠标释放”事件
public boolean mouseMove (
    Event e,                //Event object
    int x,                  //x coordinate
    int y)                  //y coordinate
    处理“鼠标移动”事件，当移动鼠标时会产生“鼠标移动”事件
public boolean mouseExit (
    Event e,                //Event object
    int x,                  //x coordinate
    int y,                  //y coordinate
    处理“鼠标离开”事件，当鼠标指针离开一个组件的区域时会产生“鼠标离开”事件
public boolean mouseEnter (
    Event e,                //Event object
    int x,                  //x coordinate
    int y,                  //y coordinate
    处理“鼠标进入”事件，当鼠标指针进入一个组件的区域时会产生“鼠标进入”事件
public boolean mouseDrag (
    Event e,                //Event object
    int x,                  //x coordinate
    int y)                  //y coordinate
    处理一个“鼠标拖动”事件，当按住鼠标按键并移动鼠标时会产生“鼠标拖动”事件

```

图 10.23 Component 类的鼠标方法

每个鼠标方法都带有 3 个参数——事件名称、x 坐标和 y 坐标，x 坐标和 y 坐标表示事件发生的位置。当按下鼠标按键时，就调用 `mouseDown` 方法（如图 10.24 所示）；当释放鼠标按键时，就调用 `mouseUp` 方法（如图 10.24 所示）；当移动鼠标时，就调用 `mouseMove` 方法（如图 10.25 所示）；当鼠标指针进入某一组件的边界时，就调用 `mouseEnter` 方法（如图 10.26 所示）。鼠标指针是屏幕上的一个小箭头，表示鼠标的位置。当鼠标指针离开某一组件的边界时，就调用 `mouseExit` 方法（如图 10.26 所示）；当按住鼠标按键并移动鼠标时（该动作称为点击和拖动），就调用 `mouseDrag` 方法（如图 10.27 所示）。`mouseDrag` 事件在 `mouseDown` 事件之后和 `mouseUp` 事件之前发生。

图 10.24 的程序重写了 `mouseDown` 方法和 `mouseUp` 方法。当用户在某个特殊位置按下鼠标按键时，就会产生 `mouseDown` 事件，此时鼠标的坐标将显示在状态栏中。当释放鼠标按键时，在窗体中显示该坐标值。

在程序中，产生 `mouseUp` 事件的位置坐标保存在变量 `xUp` 和 `yUp` 中，布尔变量 `first` 表示鼠标按键是否是第一次释放。

在 `init` 方法中，将 `first` 设置为 `true`，表示 applet 是第一次运行。`paint` 方法中的 `if` 语句可以使 applet 在首次运行时不执行下面的语句：

```

String s = " MouseUp at ";
g.setFont ( f );

```

```
s+= " (" + xUp + ", " + yUp + ")!";
g.drawString ( s, xUp, yUp );
```

当产生 mouseDown 事件时, 坐标将显示在状态栏中。释放鼠标按键会产生 mouseUp 事件。将变量 first 设置成 false, 从而使 paint 方法中的语句能够执行; 坐标值保存在变量 xUp 和 yUp 中; 然后调用 repaint 方法, 即可自动调用 paint 方法。这样, mouseUp 的坐标就显示在窗体上。

```
1 // Fig. 10.24: MouseUpAndDown.java
2 // Demonstrating methods mouseUp and mouseDown.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MouseUpAndDown extends Applet {
7     private int xUp, yUp;
8     private boolean first;
9     private Font f;
10
11     public void init()
12     {
13         // disable statements in paint initially
14         first = true;
15
16         f = new Font( "TimesRoman", Font.BOLD, 14 );
17     }
18
19     public void paint( Graphics g )
20     {
21         // these statements will not execute
22         // on the first call to paint
23         if ( first == false ) {
24             String s = "MouseUp at ";
25             g.setFont( f );
26             s += "(" + xUp + ", " + yUp + ")!";
27             g.drawString( s, xUp, yUp );
28         }
29     }
30
31     public boolean mouseDown( Event e, int x, int y )
32     {
33         showStatus( "mouseDown at (" + x + ", " + y + ")" );
34         return true; // event has been handled
35     }
36
37     public boolean mouseUp( Event e, int x, int y )
38     {
39         first = false; // enable statements in paint
40         xUp = x;
41         yUp = y;
42         repaint();
43         return true; // event has been handled
44     }
45 }
```

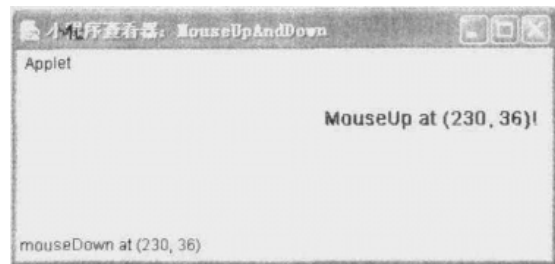


图 10.24 mouseUp 事件和 mouseDown 事件

图 10.25 的程序说明了 mouseMove 事件，该程序由一个不带 GUI 组件的 applet 组成。这里重写了 mouseMove 事件，它的功能是在状态栏中显示鼠标的坐标。坐标值随着鼠标的移动而不断发生变化。当鼠标指针移出该 applet 时，即使鼠标移动，也不会再产生 mouseMove 事件。

```

1 // Fig. 10.25: MoveTheMouse.java
2 // Demonstrating method mouseMove.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MoveTheMouse extends Applet {
7
8     public boolean mouseMove( Event e, int x, int y )
9     {
10         showStatus( "mouse at (" + x + ", " + y + ")" );
11         return true; // event has been handled
12     }
13 }

```

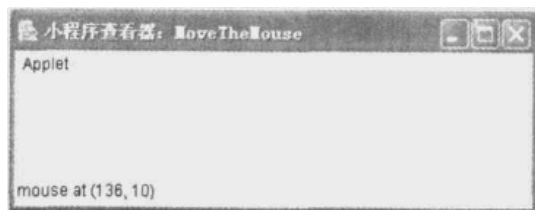


图 10.25 mouseMove 方法

图 10.26 的程序说明了 mouseEnter 事件和 mouseExit 事件。当鼠标进入 applet 时，就会产生 mouseEnter 事件；当鼠标离开 applet 时，则会产生 mouseExit 事件。当鼠标指针进入 applet 区域时，将会立即触发 mouseEnter 事件；当鼠标指针离开 applet 区域时，则会立即触发 mouseExit 事件。

```

1 // Fig. 10.26: MouseEnterExit.java
2 // Demonstrating methods mouseEnter and mouseExit.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MouseEnterExit extends Applet {
7     private Label l;
8     private List t;
9
10    public void init()
11    {
12        t = new List( 3, false );

```



```

13         l = new Label( "Measure in: " );
14
15         t.addItem( "Metric system" );
16         t.addItem( "Marak system" );
17         t.addItem( "English system" );
18
19         add( l );
20         add( t );
21     }
22
23     public boolean mouseEnter( Event e, int x, int y )
24     {
25         showStatus( "Entered area of applet" );
26         return true;    // event has been handled
27     }
28
29     public boolean mouseExit( Event e, int x, int y )
30     {
31         showStatus( "Exited area of applet" );
32         return true;    // event has been handled
33     }
34 }

```

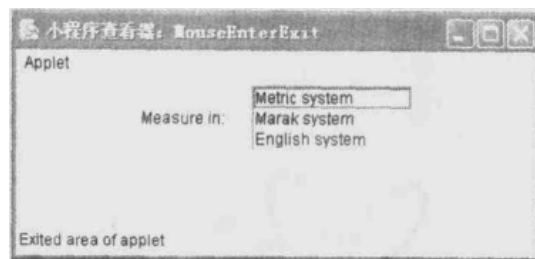


图 10.26 mouseEnter 事件和 mouseExit 事件

图 10.27 的程序利用 mouseDrag 事件创建了一个简单的绘图程序,用户可以通过鼠标的点击和拖动来进行绘图。

```

1  // Fig. 10.27: Drag.java
2  // Demonstrating method mouseDrag.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class Drag extends Applet {
7      private int xValue, yValue;
8      private boolean firstTime;
9
10     public void init()
11     {
12         // first running of program
13         firstTime = true;
14     }
15
16     public void paint( Graphics g )
17     {
18         // do not draw the first time
19         if ( !firstTime )

```

```

20         g.fillOval( xValue, yValue, 4, 4 );
21     }
22
23     // override Component class update
24     public void update( Graphics g )
25     {
26         // do not clear background
27         // only call paint
28         paint( g );
29     }
30
31     public boolean mouseDrag( Event e, int x, int y )
32     {
33         xValue = x;
34         yValue = y;
35
36         // enable drawing
37         firstTime = false;
38
39         repaint();      // call repaint
40         showStatus( "Event: mouseDrag" );
41
42         return true;    // event handled
43     }
44 }

```



图 10.27 mouseDrag 方法

变量 `xValue` 和 `yValue` 中保存着产生 `mouseDrag` 事件时的坐标值。布尔变量 `firstTime` 用于标识 `paint` 方法是否是第一次执行。在 `init` 方法中，将 `firstTime` 的值设置为 `true`，这样可避免程序一执行就在 (0,0) 的位置上画出一个填充的椭圆。在这个程序中重写了 `paint`、`update` 和 `mouseDrag` 方法。

在 `mouseDrag` 方法中，将变量 `xValue` 和 `yValue` 分别赋值为 `x` 和 `y`，并将变量 `firstTime` 设置为 `false`。现在就可以开始画图了，因为 applet 将在 `mouseDrag` 事件发生之前适当地进行初始化。接着调用 `repaint` 方法，即调用 `paint` 和 `update` 方法。

程序中重写了 `update` 方法，该方法将直接调用 `paint` 方法，而没有清除 applet 的背景。这样，用户绘制的所有内容都会在屏幕上保留下来。如果没有重写 `update`，那么只有最后画出的图形会显示在 applet 上。`paint` 方法的功能是在坐标 (`xValue`, `yValue`) 的位置上画一个椭圆，椭圆的宽度和高度都为 4。

图 10.28 中描述了 `Event` 类的鼠标常量，图 10.29 的程序使用这些常量来判断鼠标事件。每当发生一个事件，applet 都会向一个列表中添加一个字符串，表示发生了哪一个鼠标事件。

| Component类的鼠标常量                                  |                  |
|--------------------------------------------------|------------------|
| <code>public final static int MOUSE_DOWN</code>  | 整数常量, 代表“鼠标按下”事件 |
| <code>public final static int MOUSE_UP</code>    | 整数常量, 代表“鼠标释放”事件 |
| <code>public final static int MOUSE_MOVE</code>  | 整数常量, 代表“鼠标移动”事件 |
| <code>public final static int MOUSE_EXIT</code>  | 整数常量, 代表“鼠标离开”事件 |
| <code>public final static int MOUSE_ENTER</code> | 整数常量, 代表“鼠标进入”事件 |
| <code>public final static int MOUSE_DRAG</code>  | 整数常量, 代表“鼠标拖动”事件 |

图 10.28 Component类的鼠标常量

```

1  // Fig. 10.29: MouseEvents.java
2  // Demonstrating mouse Event constants.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class MouseEvents extends Applet {
7      private List eventList;
8
9      public void init()
10     {
11         eventList = new List( 6, false );
12         add( eventList );
13     }
14
15     public boolean handleEvent( Event e )
16     {
17         // determine which mouse event occurred
18         switch ( e.id ) {
19             case Event.MOUSE_UP:
20                 eventList.addItem( "Mouse up" );
21                 return true;
22
23             case Event.MOUSE_DOWN:
24                 eventList.addItem( "Mouse down" );
25                 return true;
26
27             case Event.MOUSE_MOVE:
28                 showStatus( "Mouse move" );
29                 return true;
30
31             case Event.MOUSE_ENTER:
32                 eventList.addItem( "Mouse enter" );
33                 return true;
34
35             case Event.MOUSE_EXIT:
36                 eventList.addItem( "Mouse exit" );
37                 return true;
38
39             case Event.MOUSE_DRAG:
40                 showStatus( "Mouse drag" );
41                 return true;
42         }
43
44         // not one of our mouse events

```

```

45      showStatus( "Not one of the mouse events!" );
46      return true; // done processing
47  }
48  }

```

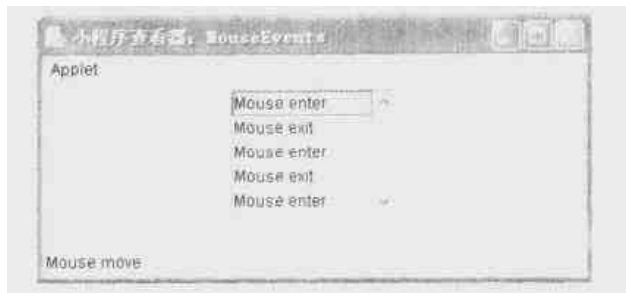


图 10.29 说明鼠标事件常量

该程序重写了 `Component` 类的 `handleEvent` 方法，`handleEvent` 方法带有一个 `Event` 参数，并返回一个布尔值。`handleEvent` 的默认功能是识别不同的事件（如鼠标移动事件、鼠标离开事件等），并调用相应的事件处理程序（如 `mouseMove` 方法、`mouseExit` 方法等）。每个事件都使用一个惟一的序号标识自己，这个序号保存在 `Event` 的实例变量 `id` 中。我们将在第 11 章更详细地介绍 `handleEvent` 方法。

在 `handleEvent` 方法中，首先通过 `Event` 实例变量 `id` 来判断是哪一個鼠标事件常量。当识别出一个鼠标事件后，就将相应的字符串添加到列表中或写到状态栏上，然后返回 `true` 值，表示该事件已被处理。在本例中，没有处理非鼠标事件，只是返回一个 `true` 值，表示没有对非鼠标事件进行更多的处理。

## 10.10 键盘事件

当按下和释放键盘按键时会产生键盘事件，在 `Component` 类中，用于处理键盘事件的方法包括 `keyUP` 和 `keyDown`，如图 10.30 所示。

| Component 类的键盘方法                      |                                              |
|---------------------------------------|----------------------------------------------|
| <code>public boolean keyDown (</code> |                                              |
| <code>Event e,</code>                 | <code>//event</code>                         |
| <code>int key )</code>                | <code>//Unicode value of key pressed</code>  |
| 处理“按键按下”事件。当一个按键按下时，会产生“按键按下”事件       |                                              |
| <code>public boolean keyUp (</code>   |                                              |
| <code>Event e,</code>                 | <code>//event</code>                         |
| <code>int key )</code>                | <code>//Unicode value of key released</code> |
| 处理“按键释放”事件。当一个按键释放时，会产生“按键释放”事件       |                                              |

图 10.30 Component 类的键盘方法

当按下一个按键时，会产生 `keyDown` 事件；当释放一个按键时，则产生 `keyUp` 事件。图 10.31 的程序说明了 `keyDown` 和 `keyUp` 的用法，当按下或释放一个按键时，相应的字符和动作信息（按下或释放该按键）会显示在状态栏中；同时，按下的按键字符还将显示在 applet 中。

`Event` 实例变量 `key` 包含按下或释放的按键的内部码值，这个内部码值将转换成字符显示出来。

在 `Event` 类中，还定义了一些表示特殊按键的键盘常量，比如 `Home` 和 `End` 键，在图 10.32 中列出了这些 `Event` 类常量。

```

1  // Fig. 10.31: Key.java
2  // Demonstrating methods keyUp and keyDown.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class Key extends Applet {
7      private Font f;
8      private String letter;
9      private boolean first;
10
11     public void init()
12     {
13         f = new Font( "Courier", Font.BOLD, 72 );
14         first = true;
15     }
16
17     public void paint( Graphics g )
18     {
19         g.setFont( f );
20
21         if ( !first )
22             g.drawString( letter, 75, 70 );
23     }
24
25     public boolean keyDown( Event e, int key )
26     {
27         showStatus( "keyDown: the " + ( char ) key +
28                     " was pressed." );
29
30         letter = String.valueOf( ( char ) key );
31         first = false;
32         repaint();
33
34         return true;    // event has been handled
35     }
36
37     public boolean keyUp( Event e, int key )
38     {
39         showStatus( "keyUp: the " + ( char ) key +
40                     " was released." );
41
42         return true;    // event has been handled
43     }
44 }

```



图 10.31 keyDown 方法和 keyUp 方法

| Event 类的特殊按键常量                             |                                     |
|--------------------------------------------|-------------------------------------|
| <code>public final static int UP</code>    | 整数常量, 表示向上方向键                       |
| <code>public final static int DOWN</code>  | 整数常量, 表示向下方向键                       |
| <code>public final static int LEFT</code>  | 整数常量, 表示向左方向键                       |
| <code>public final static int RIGHT</code> | 整数常量, 表示向右方向键                       |
| <code>public final static int END</code>   | 整数常量, 表示 END 键                      |
| <code>public final static int HOME</code>  | 整数常量, 表示 HOME 键                     |
| <code>public final static int PGDN</code>  | 整数常量, 表示向下翻页键                       |
| <code>public final static int PGUP</code>  | 整数常量, 表示向上翻页键                       |
| <code>public final static int F1</code>    | 整数常量, 表示 F1 键, 还有表示 F2 ~ F12 键的整数常量 |

图 10.32 Event 类的特殊按键常量

图 10.33 程序的功能是判断所按下的键是否为特殊按键之一(如图 10.32 中所列)。每当按下一个特殊按键时, 相应的选项就会添加到列表框中, 从而标识按下的是哪一个键。

该程序重写了 Component 类的 `handleEvent` 方法。程序中通过一个 `switch` 结构和 Event 的实例变量 `key` 来判断按下的是哪一个特殊按键。然后, 在文本字段中显示相应的信息。

```

1 // Fig. 10.33: KeyEvents.java
2 // Demonstrating keyboard Event constants.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class KeyEvents extends Applet {
7     private TextField t;
8
9     public void init()
10    {
11        t = new TextField( 25 );
12        t.setEditable( false );
13        add( t );
14    }
15
16    public boolean handleEvent( Event e )
17    {
18        // determine which key event occurred
19        switch ( e.key ) {
20            case Event.PGUP:
21                t.setText( "PageUp key" );
22                return true;
23
24            case Event.PGDN:
25                t.setText( "PageDown key" );
26                return true;

```

```

27
28     case Event.END:
29         t.setText( "End key" );
30         return true;
31
32     case Event.RIGHT:
33         t.setText( "Right arrow key");
34         return true;
35
36     case Event.LEFT:
37         t.setText( "Left arrow key");
38         return true;
39
40     case Event.UP:
41         t.setText( "Up arrow key" );
42         return true;
43
44     case Event.DOWN:
45         t.setText( "Down arrow key" );
46         return true;
47
48     case Event.HOME:
49         t.setText( "Home key" );
50         return true;
51
52     case Event.F1: case Event.F2: case Event.F3:
53     case Event.F4: case Event.F5: case Event.F6:
54     case Event.F7: case Event.F8: case Event.F9:
55     case Event.F10: case Event.F11: case Event.F12:
56         t.setText( "A function key ( F1 - F12 )" );
57         return true;
58     }
59
60     // not one of our key events
61     t.setText( "Not a special key!" );
62     return true; // done processing
63 }
64 }

```

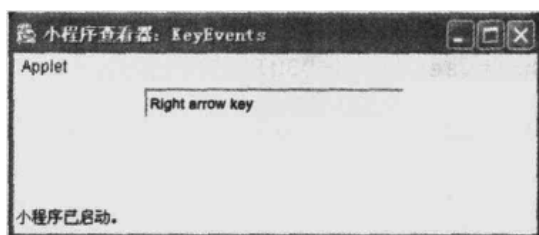


图 10.33 特殊按键常量

Ctrl键、元(meta)键和Shift键的状态也可以进行判断。图10.34给出了Event类的controlDown方法、metaDown方法和shiftDown方法。

图10.35的程序说明了controlDown方法、metaDown方法和shiftDown方法的用法。3个不可编辑的文本字段用于显示每个按键的名字，Ctrl键、Shift键和元键可以进行任意组合。

---

**Event 方法 controlDown、metaDown 和 shiftDown**


---

```
public boolean controlDown()
    如果按下 Ctrl 键, 则返回 true 值
public boolean metaDown()
    如果按下元键, 则返回 true 值
public boolean shiftDown()
    如果按下 Shift 键, 则返回 true 值
```

---

图 10.34 测试 Ctrl 键、元键和 Shift 键状态的方法

```
1 // Fig. 10.35: MetaShiftCtrl.java
2 // Demonstrating methods controlDown, metaDown and shiftDown.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MetaShiftCtrl extends Applet {
7     private TextField meta, shift, ctrl;
8
9     public void init()
10    {
11        meta = new TextField( 10 );
12        shift = new TextField( 10 );
13        ctrl = new TextField( 10 );
14
15        meta.setEditable( false );
16        ctrl.setEditable( false );
17        shift.setEditable( false );
18
19        add( meta );
20        add( shift );
21        add( ctrl );
22    }
23
24    public boolean handleEvent( Event e )
25    {
26        if ( e.metaDown() )
27            meta.setText( "META" );
28
29        if ( e.controlDown() )
30            ctrl.setText( "CTRL" );
31
32        if ( e.shiftDown() )
33            shift.setText( "SHIFT" );
34
35        return true; //event has been handled
36    }
37 }
```

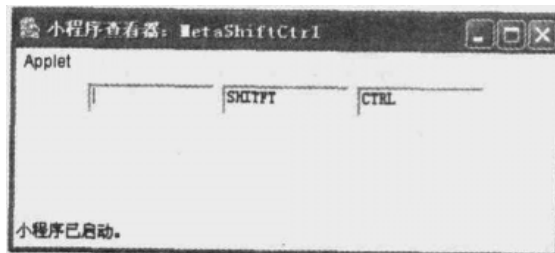


图 10.35 说明 controlDown、metaDown 和 shiftDown 方法



## 10.11 FlowLayout 布局管理器

复杂的 GUI（如图 10.1 所示）要求其中的每一个组件都放置到精确的位置上。它们通常由多个面板组成，每个面板上的组件都按指定的顺序排列。布局管理器用于安排容器上的组件，图 10.36 中列出了本章将要讨论的布局管理器。在下一章，我们将讨论更多的布局管理器。

| 布局管理器        | 说明                                    |
|--------------|---------------------------------------|
| FlowLayout   | 默认用于 applet 和面板，将各个组件按添加的顺序依次排列（从左向右） |
| BorderLayout | 将组件排列到 5 个区域：北区、南区、东区、西区和中区           |
| GridLayout   | 将组件按行和列进行排列                           |

图 10.36 布局管理器

前面的例子已经用到了 FlowLayout 布局管理器，默认情况下，FlowLayout 用于面板和 applet。FlowLayout 类是从 Object 类继承而来的，并提供了 LayoutManager 接口。LayoutManager 接口定义了一些方法，每个布局管理器都可以使用这些方法来安排容器中的组件。

FlowLayout 是最基本的布局管理器。各个 GUI 组件按照它们添加到容器上的顺序从左向右进行排列。当组件排列到容器边界时，就继续从下一行开始。图 10.37 中列出了 FlowLayout 类的构造函数和常量。

| FlowLayout 的构造函数和常量                |                                                                                                                          |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| public final static int CENTER     | 常量值，表示组件在 FlowLayout 中居中对齐，组件默认为居中对齐                                                                                     |
| public final static int LEFT       | 常量值，表示组件在 FlowLayout 中左对齐                                                                                                |
| public final static int RIGHT      | 常量值，表示组件在 FlowLayout 中右对齐                                                                                                |
| public FlowLayout ()               | 创建一个居中对齐的 FlowLayout                                                                                                     |
| public FlowLayout( int alignment ) | //component alignment<br>创建一个按指定方式对齐的 FlowLayout，参数 alignment 的值可以是 FlowLayout.RIGHT、FlowLayout.LEFT 或 FlowLayout.CENTER |
| public FlowLayout (                |                                                                                                                          |
| int alignment ,                    | //Component alignment                                                                                                    |
| int horizontal_gap,                | //horizontal distance between components                                                                                 |
| int vertical_gap )                 | //vertical distance between components                                                                                   |
|                                    | 创建一个按指定方式对齐的 FlowLayout，参数 alignment 的值可以是 FlowLayout.RIGHT、FlowLayout.LEFT 和 FlowLayout.CENTER，组件之间的距离（像素值）由后两个参数指定     |

图 10.37 FlowLayout 的构造函数和常量

图 10.38 的程序创建了 4 个 TextField 对象，并默认地使用 FlowLayout 布局管理器将它们添加到 applet 中，各组件自动在 applet 上居中对齐。

```

1    // Fig. 10.38: FlowDefault.java
2    // Demonstrating FlowLayout default alignment.
3    import java.applet.Applet;
4    import java.awt.*;
5
6    public class FlowDefault extends Applet {

```

```

7     private TextField t1, t2, t3, t4;
8
9     public void init()
10    {
11        // use the default layout manager
12        // for the applet which is FlowLayout
13
14        t1 = new TextField( "Default " );
15        t1.setEditable( false );
16        add( t1 );
17
18        t2 = new TextField( "Default " );
19        t2.setEditable( false );
20        add( t2 );
21
22        t3 = new TextField( "Default " );
23        t3.setEditable( false );
24        add( t3 );
25
26        t4 = new TextField( "Default " );
27        t4.setEditable( false );
28        add( t4 );
29    }
30 }

```

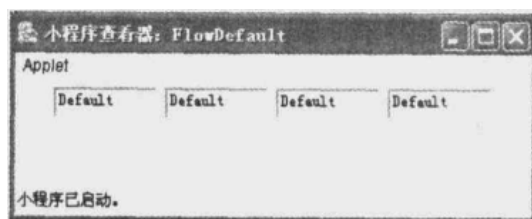


图 10.38 FlowLayout 默认对齐方式

图 10.39 的程序创建了 4 个 TextField 对象，并将它们添加到 applet 中，各个组件在 FlowLayout 中居中对齐。该程序所创建的界面与图 10.38 的一样。

容器的布局是通过 Container 类的 setLayout 方法设置的。下面的语句用于创建一个新的居中对齐的 FlowLayout 布局管理器，并使用 setLayout 方法进行设置：

```
setLayout ( newFlowLayout ( FlowLayout.CENTER ) );
```

当向 applet 上添加组件时，这些组件的位置由 FlowLayout 布局管理器来安排。

```

1    // Fig. 10.39: FlowCenter.java
2    // Demonstrating FlowLayout center alignment.
3    import java.applet.Applet;
4    import java.awt.*;
5
6    public class FlowCenter extends Applet {
7        private TextField t1, t2, t3, t4;
8
9        public void init()
10       {
11           // set layout
12           setLayout( new FlowLayout( FlowLayout.CENTER ) );
13       }

```

```

14      t1 = new TextField( "Center " );
15      t1.setEditable( false );
16      add( t1 );
17
18      t2 = new TextField( "Center " );
19      t2.setEditable( false );
20      add( t2 );
21
22      t3 = new TextField( "Center " );
23      t3.setEditable( false );
24      add( t3 );
25
26      t4 = new TextField( "Center " );
27      t4.setEditable( false );
28      add( t4 );
29  }
30  }

```

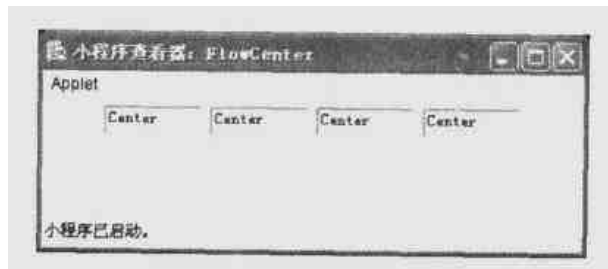


图 10.39 FlowLayout 的居中对齐方式

图 10.40 的程序创建了 4 个 TextField 对象, 并将它们添加到 applet 中, 这些组件在 FlowLayout 中是右对齐的。其中, 使用 FlowLayout.RIGHT 常量对 FlowLayout 对象进行初始化。

```

1  // Fig. 10.40: FlowRight.java
2  // Demonstrating FlowLayout right alignment.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class FlowRight extends Applet {
7      private TextField t1, t2, t3, t4;
8
9      public void init()
10     {
11         // set layout
12         setLayout( new FlowLayout( FlowLayout.RIGHT ) );
13
14         t1 = new TextField( "Right " );
15         t1.setEditable( false );
16         add( t1 );
17
18         t2 = new TextField( "Right " );
19         t2.setEditable( false );
20         add( t2 );
21
22         t3 = new TextField( "Right " );
23         t3.setEditable( false );
24         add( t3 );
25
26         t4 = new TextField( "Right " );
27         t4.setEditable( false );

```

```

28         add( t4 );
29     }
30 }

```

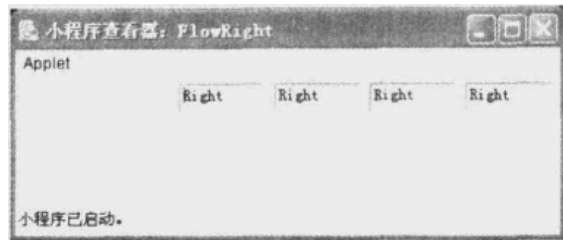


图 10.40 右对齐的 FlowLayout

图 10.41 的程序创建了 4 个 TextField 对象，并将它们添加到 applet 中，这些组件在 FlowLayout 中是左对齐的。其中，使用 FlowLayout.LEFT 常量初始化 FlowLayout 对象。

```

1  // Fig. 10.41: FlowLeft.java
2  // Demonstrating FlowLayout left alignment.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class FlowLeft extends Applet {
7      private TextField t1, t2, t3, t4;
8
9      public void init()
10     {
11         // set layout
12         setLayout( new FlowLayout( FlowLayout.LEFT ) );
13
14         t1 = new TextField( "Left " );
15         t1.setEditable( false );
16         add( t1 );
17
18         t2 = new TextField( "Left " );
19         t2.setEditable( false );
20         add( t2 );
21
22         t3 = new TextField( "Left " );
23         t3.setEditable( false );
24         add( t3 );
25
26         t4 = new TextField( "Left " );
27         t4.setEditable( false );
28         add( t4 );
29     }
30 }

```

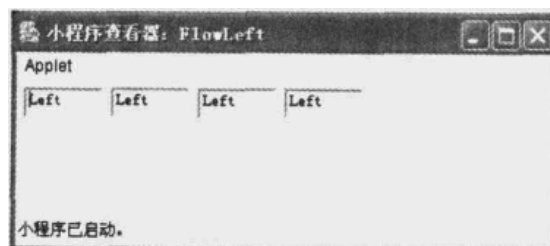


图 10.41 左对齐的 FlowLayout

图10.42的程序创建了3个Panel对象,在每个Panel中又添加了两个Textfield对象。第一个面板使用的是右对齐的FlowLayout,组件之间的水平距离为0(像素点),垂直距离为10。第二个面板使用的是左对齐的FlowLayout,组件之间的水平距离为10,垂直距离为0。第三个面板使用的是居中对齐的FlowLayout,组件之间的水平距离为10,垂直距离为10。这里修改了每个面板的背景颜色,以便突出显示。

下面的语句是将p1的布局管理器设置为右对齐,并将每个组件之间的水平距离设置为0,垂直距离设置为10:

```
p1.setLayout ( newFlowLayout ( FlowLayout.RIGHT , 0, 10 ) );
```

可以使用同样的方法设置p2和p3。注意,这些面板在applet上是居中对齐的,因为没有修改默认值。

---

```
1 // Fig. 10.42: Flow.java
2 // Demonstrating FlowLayout component spacing.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class Flow extends Applet {
7     private TextField t1, t2, t3, t4, t5, t6;
8     private Panel p1, p2, p3;
9
10    public void init()
11    {
12        p1 = new Panel();
13        p2 = new Panel();
14        p3 = new Panel();
15
16        // set layout for panel p1
17        p1.setLayout( new FlowLayout( FlowLayout.RIGHT,
18                                     0, 10 ) );
19
20        t1 = new TextField( "Text " );
21        t2 = new TextField( "Field " );
22        p1.add( t1 );
23        p1.add( t2 );
24
25        // set layout for panel p2
26        p2.setLayout( new FlowLayout( FlowLayout.LEFT,
27                                     10, 0 ) );
28
29        t3 = new TextField( "Text " );
30        t4 = new TextField( "Field " );
31        p2.add( t3 );
32        p2.add( t4 );
33
34        p3.setLayout( new FlowLayout( FlowLayout.CENTER,
35                                     10, 10 ) );
36
37        t5 = new TextField( "Text " );
38        t6 = new TextField( "Field " );
39        p3.add( t5 );
40        p3.add( t6 );
41
```

```

42         // change background colors of panels
43         p1.setBackground( Color.pink );
44         p2.setBackground( Color.yellow );
45         p3.setBackground( Color.cyan );
46         add( p1 );
47         add( p2 );
48         add( p3 );
49     }
50 }

```

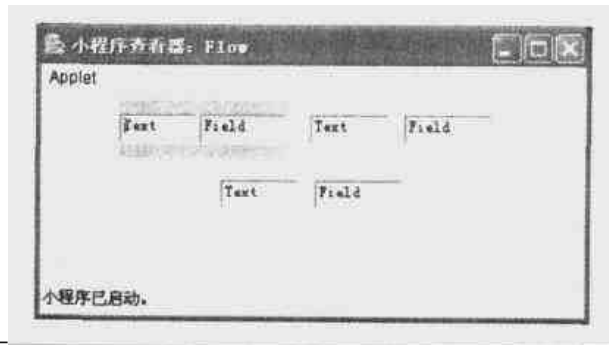


图 10.42 说明组件之间的距离

## 10.12 BorderLayout 布局管理器

BorderLayout 布局管理器将组件按 5 个区域安排：北区、南区、东区、西区和中区（北区对应于容器的顶部）。BorderLayout 是从 Object 类继承而来的，并提供了 LayoutManager 接口。图 10.43 中列出了 BorderLayout 类的构造函数。

### BorderLayout 类的构造函数

```

public BorderLayout ()
    创建一个 BorderLayout 布局管理器

public BorderLayout( int horizontalGap,int verticalGap )
    创建一个 BorderLayout 布局管理器，组件之间的水平距离和垂直距离分别由 horizontalGap 和 verticalGap 参数确定

```

图 10.43 BorderLayout 类的构造函数

图 10.44 的程序使用 5 个按钮组件说明了 BorderLayout 布局管理器的用法。setLayout 方法用于设置 BorderLayout，注意 add 参数的差别。例如，下面的语句是将按钮 southButton 放置在南区：

```
add ( " South " ,southButton );
```

因为组件放置的位置必须由第一个参数指定，所以可以按任何顺序添加组件。

```

1  // Fig. 10.44: Border.java
2  // Demonstrating BorderLayout.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class Border extends Applet {
7      private Button centerButton, eastButton, northButton,
8                  westButton, southButton;
9

```

```

10     public void init()
11     {
12         // instantiate button objects
13         centerButton = new Button( "center Button" );
14         westButton = new Button( "west Button" );
15         eastButton = new Button( "east Button" );
16         southButton = new Button( "south Button" );
17         northButton = new Button( "north Button" );
18
19         // set layout to border layout
20         setLayout( new BorderLayout() );
21
22         // order not important
23         add( "South", southButton );           // South position
24         add( "North", northButton );          // North position
25         add( "East", eastButton );            // East position
26         add( "Center", centerButton );        // Center position
27         add( "West", westButton );            // West position
28     }
29 }

```

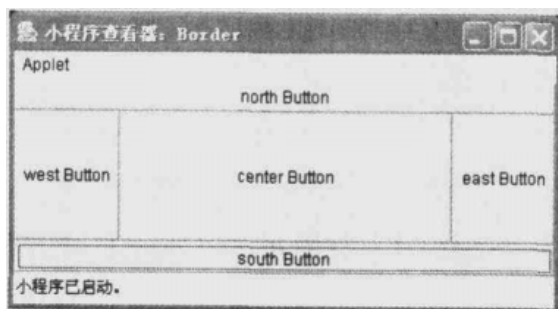


图 10.44 BorderLayout 布局管理器

首先设置北区和南区的大小,它们沿着容器的边缘排列。接着设置东区和西区的大小,最后再设置中区,中区将由北区、南区、东区和西区包围。

在 BorderLayout 中,最多可以包含 5 个组件——每个位置包含一个。如果未使用某个位置(除了中区),该位置就由其他组件占用。例如,如果未使用北区,那么东区、中区和西区的组件就会向上扩展到容器的边缘。

图 10.45 的程序使用了一个带有 4 个组件的 BorderLayout 布局管理器。东区没有使用,中区的组件向右扩展,并占用了东区。

```

1 // Fig. 10.45: Border2.java
2 // Demonstrating BorderLayout with four components.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class Border2 extends Applet {
7     private Button centerButton, northButton,
8         westButton, southButton;
9
10    public void init()
11    {
12        // instantiate button objects
13        centerButton = new Button( "center Button" );

```

```

14     westButton = new Button( "west Button" );
15     southButton = new Button( "south Button" );
16     northButton = new Button( "north Button" );
17
18     // set layout to border layout
19     setLayout( new BorderLayout() );
20
21     // order not important
22     add( "South", southButton );           // South position
23     add( "North", northButton );          // North position
24     add( "Center", centerButton );        // Center position
25     add( "West", westButton );            // West position
26 }
27 }

```

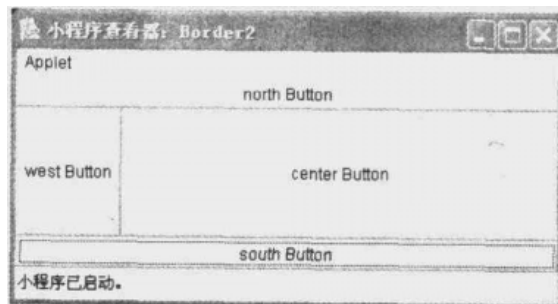


图 10.45 带有 4 个组件的 BorderLayout

图 10.46 的程序使用了一个带有 5 个按钮的 BorderLayout 布局管理器，每个按钮之间的水平和垂直距离分别为 20 和 15 个像素。

下面的语句用于将 applet 的布局管理器设置成 BorderLayout，每个组件之间的水平和垂直距离分别为 20 和 15 个像素：

```
setLayout( new BorderLayout( 20, 15 ) );
```

#### 常见编程错误 10.10

在 BorderLayout 中，如果将 add 方法的第一个参数写成“north”、“south”、“east”、“west”或“center”，那么运行时将会产生逻辑错误。在某些情况下，组件可能根本显示不出来，这是因为表示方位的名字必须以大写字母开头（如“North”）。

```

1  // Fig. 10.46: Border3.java
2  // Demonstrating BorderLayout component spacing.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class Border3 extends Applet {
7      private Button centerButton, eastButton, northButton,
8                  westButton, southButton;
9
10     public void init()
11     {
12         // instantiate button objects
13         centerButton = new Button( "center Button" );
14         westButton = new Button( "west Button" );
15         eastButton = new Button( "east Button" );
16         southButton = new Button( "south Button" );

```



```

17         northButton = new Button( "north Button");
18
19         // set layout to border layout
20         setLayout( new BorderLayout( 20, 15 ) );
21
22         // order not important
23         add( "South", southButton );           // South position
24         add( "North", northButton );           // North position
25         add( "East", eastButton );             // East position
26         add( "Center", centerButton );         // Center position
27         add( "West", westButton );             // West position
28     }
29 }

```

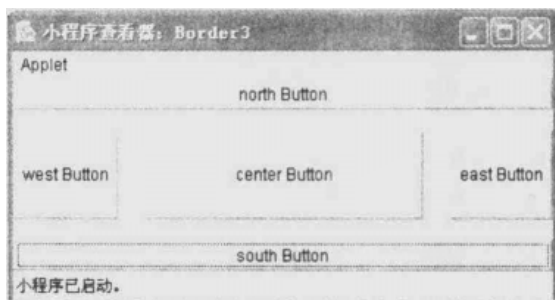


图 10.46 BorderLayout 的组件之间的距离

## 10.13 GridLayout 布局管理器

GridLayout 布局管理器将容器划分成网格, 这样各个组件就可以按行列放置到每个网格中, 每个组件的大小都是一样的。GridLayout 类是从 Object 类直接继承而来的, 并提供 LayoutManager 接口。在向 GridLayout 添加组件时, 其顺序是从网格的左上角开始, 从左向右排列, 直到排满一行, 然后再从下一行开始从左向右排列, 依次进行。图 10.47 中列出了 GridLayout 类的构造函数。

### GridLayout 类的构造函数

```

public GridLayout (
    int rows ,                //number of rows
    int columns)              // number of columns
构造一个 GridLayout 布局管理器, 行数和列数分别为 rows 和 columns

public GridLayout (
    int rows ,                // number of rows
    int columns               // number of columns
    int h,                   // horizontal spacing between components
    int v)                   // vertical spacing between components
构造一个 GridLayout, 每个组件之间的水平距离和垂直距离分别为 h 个像素和 v 个像素

```

图 10.47 GridLayout 构造函数

图 10.48 的程序使用 5 个按钮说明了 GridLayout 布局管理器的用法。按钮对象在网格中排成两行三列。setLayout 方法用于设置 GridLayout 管理器。第一个组件将添加到第一行的第一列上。第二个组件将添加第一行的第二列上, 依次类推。因为一共只有 5 个组件, 所以第二行的第三列为空。

```

1 // Fig. 10.48: Grid.java
2 // Demonstrating GridLayout.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class Grid extends Applet {
7     private Button button1, button2, button3,
8         button4, button5;
9
10    public void init()
11    {
12        // instantiate button objects
13        button1 = new Button( "one" );
14        button2 = new Button( "two" );
15        button3 = new Button( "three" );
16        button4 = new Button( "four" );
17        button5 = new Button( "five" );
18
19        // set layout to grid layout
20        setLayout( new GridLayout( 2, 3 ) );
21
22        // order is important
23        add( button1 ); // row 1 column 1
24        add( button2 ); // row 1 column 2
25        add( button3 ); // row 1 column 3
26        add( button4 ); // row 2 column 1
27        add( button5 ); // row 2 column 2
28    }
29 }

```

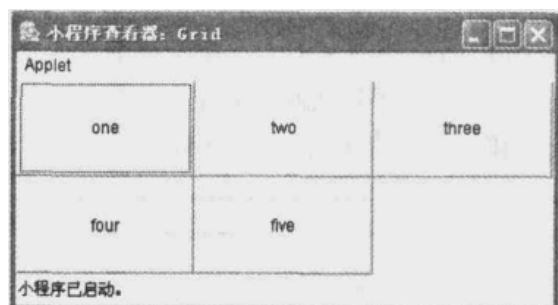


图 10.48 GridLayout 布局管理器

图 10.49 的程序创建了 6 个按钮，并将它们排成两行三列，每个组件之间的水平距离和垂直距离分别为 20 和 25 个像素。使用下面的语句设置该布局管理器：

```
setLayout ( new GridLayout( 2, 3, 20, 25 ) );
```

```

1 // Fig. 10.49: Grid2.java
2 // Demonstrating GridLayout component spacing.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class Grid2 extends Applet {
7     private Button button1, button2, button3,
8         button4, button5, button6;

```

```

9
10     public void init()
11     {
12         // instantiate button objects
13         button1 = new Button( "one" );
14         button2 = new Button( "two" );
15         button3 = new Button( "three" );
16         button4 = new Button( "four" );
17         button5 = new Button( "five" );
18         button6 = new Button( "six" );
19
20         // set layout to grid layout
21         setLayout( new GridLayout( 2, 3, 20, 25 ) );
22
23         // order is important
24         add( button1 );    // row 1 column 1
25         add( button2 );    // row 1 column 2
26         add( button3 );    // row 1 column 3
27         add( button4 );    // row 2 column 1
28         add( button5 );    // row 2 column 2
29         add( button6 );    // row 2 column 3
30     }
31 }

```

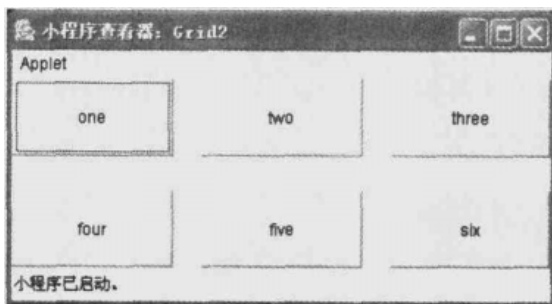


图 10.49 GridLayout 组件之间的距离

## 小结

- 图形用户界面 (GUI) 为程序提供图形化的界面, 使用户对程序能有一个直观的“视觉”和“感觉”。
- 用于创建 GUI 组件的类包含在 java.awt 软件包中。
- 标签是一个用于显示单行静态 (只读) 文本的区域。
- 容器是一个可以放置多个组件的区域, 各组件都通过 Container 类的 add 方法进行添加。当添加一个组件后, 该组件就出现在屏幕上。
- Label 的方法 getText 用于返回标签上的文本。
- Label 的方法 setText 用于设置标签上的文本。
- 按钮是一个组件, 用户点击它将会激活一个事件。
- 撇压式按钮由 Button 类创建, 按钮面上的文本称为按钮标签。
- GUI 是由事件驱动的。事件就是异步动作, 比如用户移动鼠标、点击鼠标等。事件由窗口系统发送到程序。

- 事件保存在 Event 对象中。Event 类是 java.awt 软件包的一部分。在点击一个按钮时，就会调用 action 方法，action 方法返回一个布尔值，并带有两个参数：Event 和 Object。参数 Event 中保存的是有关事件的特殊信息，参数 Object 中保存的是指定组件的信息，Event 的实例变量 target 用于标识产生事件的组件。对于按钮来说，Event 类的实例变量 arg 中包含的是按钮标签。
- 文本字段是一个单行显示区域，可以从键盘上接收用户输入。
- 如果 setEditable 方法的输入参数为 false，则可以将文本字段设置为只读的。
- 文本字段的 setEchoCharacter 方法带有一个单字符参数，可以将它作为某个文本字段的掩码字符。
- 对于文本字段来说，Event 的实例变量 arg 中包含的是按下回车键时文本字段中的文本。
- 选择按钮提供了一系列选项，用户可从中进行选择。选择按钮中的选项是通过 addItem 方法添加的，各选项的顺序由一个数字类型的下标表示，第一个添加的选项下标为 0。当将选择按钮添加到一个容器中时，首先将显示选择按钮的第一个选项，然后通过点击向下方向键来选择其他选项。当点击向下方向键时，会弹出一个选项列表供用户选择。对于选择按钮来说，Event 的实例变量 arg 中包含的是被选中选项的内容。
- 复选框按钮和单选按钮都是状态按钮，其值为 true/false。
- Checkbox 类的 setLabel 方法用于设置复选框的标签。复选框的值为 true 或 false——如果复选框被选中，则值为 true；若复选框未被选中，则值为 false。Checkbox 类的 getState 方法将返回一个布尔值，以表示复选框的状态。对于复选框来说，Event 的实例变量 arg 是一个布尔对象。
- 可以将复选框组合到一起，成为一组单选按钮。单选按钮由 CheckboxGroup 类和 Checkbox 类创建，CheckboxGroup 不是组件。
- Checkbox 的方法 getState 用于返回单选按钮的状态。
- 列表用于显示一系列的选项，用户可从中选择一个或多个，列表中的选项使用 addItem 方法进行添加。
- 使用鼠标双击列表选项会产生一个事件。对于列表来说，Event 类的实例变量 arg 表示被选中的选项。
- 在多选列表中，一次可以选择多个选项。若将 List 构造函数的第二个参数设置为 true，则可以创建一个多选列表。
- 若列表中的选项个数超过了列表的可见行数，则列表右侧会出现一个滚动条，列表的可见行数是由构造函数的第一个参数确定的。
- 要想从多选列表中选取选项，则必须激活外部事件，通常使用按钮来激活该事件。getSelectedItems 方法可用于返回列表中选中的选项。
- Panel 类是从 Container 类继承的，而 Applet 类又是从 Panel 类继承的。Panel 类的构造函数没有参数，而且没有重载。面板的大小由它所包含的组件来确定。
- 通常，当按下鼠标按键时，就调用 mouseDown 方法。当释放鼠标按键时，就调用 mouseUp 方法。当鼠标移动时，就调用 mouseMove 方法。当鼠标指针进入某一组件的边界时，就调用 mouseEnter 方法。当按住一个鼠标按键并移动鼠标时（这个动作就是点击和拖动），就调用 mouseDrag 方法。
- Component 的 handleEvent 方法带有一个 Event 参数，并返回一个布尔值。handleEvent 的默认功能是识别不同的事件，并调用相应的事件处理程序。每个事件都用一个惟一的序号标识自

己, 该序号保存在 Event 的实例变量 id 中。

- 当按下或释放键盘按键时, 会产生键盘事件, Component 类的 keyUp 方法和 keyDown 方法用于处理键盘事件。当按下按键时, 会引发 keyDown 事件, 而当释放按键时, 会引发 keyUp 事件。
- Event 类的实例变量 key 中包含的是所按下或释放的按键的内部码值。
- 一些特殊按键被定义成键盘常量, 比如 Home、End、PgDn、PgUp 以及功能键等。
- controlDown 方法、metaDown 方法和 shiftDown 方法分别用于判断 Ctrl 键、元键和 Shift 键的状态。
- 布局管理器用于安排容器上的组件, 面板和 applet 默认使用 FlowLayout 布局管理器。
- FlowLayout 是最基本的布局管理器, GUI 组件在容器上按从左到右的顺序逐个排列。FlowLayout 可以为右对齐、左对齐或居中对齐, 默认值为居中对齐。
- BorderLayout 布局管理器将组件排列成 5 个区域: 北区、南区、东区、西区和中区。
- setLayout 方法用于为容器设置新的布局。
- 在 BorderLayout 中最多可包含 5 个组件——每个区域包含一个组件。若未使用某个区, 那么该位置就由其他组件占用。
- GridLayout 布局管理器是将组件按行列排列, 每个组件的大小都一致。在添加组件时, 添加的顺序非常重要, 第一个添加的组件放在第一行的第一列, 第二个添加的组件放在第一行的第二列, 依次类推。

## 术语

action method    action 方法

add method    add 方法

addItem method    addItem 方法

Applet class    Applet 类

arg

attach a component to a container    向容器中添加一个组件

AWT(another/abstract windowing toolkit)    AWT(其他/抽象窗口化工具箱)

awt package    awt 软件包

BorderLayout class    BorderLayout 类

BorderLayout constructor    BorderLayout 构造函数

BorderLayout layout manager    BorderLayout 布局管理器

button    按钮

button event    按钮事件

button label    按钮标签

Button class    Button 类

Center in BorderLayout    BorderLayout 中的 Center

checkbox button    复选框按钮

Checkbox class    Checkbox 类

Checkbox constructor    Checkbox 构造函数

checkbox label    复选框标签

CheckboxGroup class    CheckboxGroup 类

CheckboxGroup constructor    CheckboxGroup 构造函数

choice button    选择按钮

Choice class    Choice 类

Choice constructor    Choice 构造函数

clicking and dragging    点击和拖动

component    组件

Component class    Component 类

container    容器

Container class    Container 类

control key    控制键

controlDown method    controlDown 方法

countItems method    countItem 方法

default text in a text field    文本字段中的默认文本

East in BorderLayout    BorderLayout 的 East

Event.DOWN

- Event.END
- Event.F1~Event.F12
- Event.HOME
- Event.LEFT
- Event.MOUSE\_DOWN
- Event.MOUSE\_DRAG
- Event.MOUSE\_ENTER
- Event.MOUSE\_EXIT
- Event.MOUSE\_MOVE
- Event.MOUSE\_UP
- Event.DCDN
- Event.PGUP
- Event.RIGHT
- Event.UP
- Event class Event 类
- event driven 事件驱动
- event handling 事件处理
- FlowLayout.CENTER
- FlowLayout.LEFT
- FlowLayout.RIGHT
- FlowLayout constructor FlowLayout 构造函数
- FlowLayout layout manager FlowLayout 布局管理器
- getAlignment method getAlignment 方法
- getLabel method getLabel 方法
- getItem method getItem 方法
- getSelectedIndex method getSelectedIndex 方法
- getSelectedItem method getSelectedItem 方法
- getSelectedItems method getSelectedItems 方法
- getState method getState 方法
- getText method getText 方法
- graphical user interface(GUI) 图形用户界面 (GUI)
- Graphics object Graphics 对象
- GridLayout class GridLayout 类
- GridLayout layout manager GridLayout 布局管理器
- GUI component GUI 组件
- handleEvent method handleEvent 方法
- id
- import java.awt.\*;
- index 索引
- instanceof
- key
- keyboard 键盘
- keyboard constants 键盘常量
- keyboard events 键盘事件
- keyDown method keyDown 方法
- keyUp method keyUp 方法
- label 标签
- Label class Label 类
- Label constructor Label 构造函数
- layout manager 布局管理器
- LayoutManager interface LayoutManager 接口
- list 列表
- List
- List class List 类
- List constructor List 构造函数
- meta key 元键
- metaDown method metaDown 方法
- mouse 鼠标
- mouse events 鼠标事件
- mouse pointer 鼠标指针
- mouseDown method mouseDown 方法
- mouseDrag method mouseDrag 方法
- mouseEnter method mouseEnter 方法
- mouseExit method mouseExit 方法
- mouseMove method mouseMove 方法
- mouseUp method mouseUp 方法
- multiple selections 多项选择
- non-static text 非静态的文本
- North in BorderLayout BorderLayout 中的 North
- paint method paint 方法
- panel 面板
- Panel class Panel 类
- Panel constructor Panel 构造函数
- push button 按钮
- radio button 单选按钮
- radio button label 单选按钮标签
- repaint method repaint 方法
- scrollbar 滚动条
- setAlignment method setAlignment 方法
- setBackground method setBackground 方法
- setEchoCharacter method setEchoCharacter 方法

|                                             |                                                      |
|---------------------------------------------|------------------------------------------------------|
| setEditable(false)                          | TextComponent class TextComponent 类                  |
| setEditable(true)                           | text field 文本字段                                      |
| setFont method setFont 方法                   | TextField class TextField 类                          |
| setLabel method setLabel 方法                 | TextField constructor TextField 构造函数                 |
| setLayout method setLayout 方法               | true/false Checkbox setting Checkbox 的 true/false 设置 |
| setText method setText 方法                   | true/false List setting List 的 true/false 设置         |
| Shift key Shift 键                           | update method update 方法                              |
| shiftDown method shiftDown 方法               | West in BorderLayout BorderLayout 中的 West            |
| South in BorderLayout BorderLayout 中的 South | widgets 窗口小部件                                        |
| state button 状态按钮                           | wildcard character(*) 通配符 (*)                        |
| static text 静态文本                            |                                                      |
| target                                      |                                                      |

## 自测练习

### 10.1 填空:

- 当移动鼠标时, 将产生一个 \_\_\_\_\_ 事件。
- 用户不能修改的文本称为 \_\_\_\_\_。
- \_\_\_\_\_ 用于安排 applet 或面板上的 GUI 组件。
- add 方法是 \_\_\_\_\_ 类的方法。
- GUI 是 \_\_\_\_\_ 的缩写。
- \_\_\_\_\_ 方法用于设置容器的布局管理器。
- 一个 mouseDrag 事件会引发一个 \_\_\_\_\_ 事件和一个 \_\_\_\_\_ 事件。

### 10.2 判断下列句子是否正确。如果不正确, 请说明原因。

- 面板的默认布局管理器是 BorderLayout。
- 当鼠标指针位于 GUI 组件的顶部时, 会产生 mouseOver 事件。
- 一个面板不能添加到另一个面板上。
- 当创建 GUI 时, 必须从 awt 软件包引入一个或多个类。
- 在带有 5 个按钮的 BorderLayout 中, 最后确定大小的组件是中区的按钮。
- 在使用 BorderLayout 时, 最多可以包含 5 个组件。

### 10.3 找出下列语句中的错误, 并进行改正。

```

a) buttonName= Button ( " Caption " );
b) Label aLabel ,Label1 ;//create references
c) textField=new TextField ( 50, "Default Text " ) ;
d) public class radioButtons extends Applet {
    RadioButton rb; //awt class RadioButton
    public void init ( )
    {
        this.rb =new RadioGroup ( ) ;
        //other statements
    }
}

```

```
e) public void init ( )  
{  
    setLayout ( new GridLayout( 9, 9 ) );  
    button1 = new Button ( " North Star " );  
    button2 = new Button ( "South Pole" );  
    add ( " North ", button1 );  
    add ( " South " , button2 );  
}
```

10.4 简要回答下列问题:

a) 解释下面语句中星号 (\*) 的含义:

```
import java.awt. * ;
```

b) 解释文本字段和标签之间的差别。

## 自测练习答案

- 10.1 a) mouseMove。b) 静态的 ( 只读 )。c) 布局管理器。d) Container。e) 图形用户界面 ( Graphical User Interface )。f) setLayout。g) mouseDown, mouseUp。
- 10.2 a) 不正确。默认的布局管理器是 FlowLayout。
- b) 不正确。产生 mouseEnter 事件。
- c) 不正确。一个面板可以添加到另一个面板上。
- d) 正确。
- e) 正确。
- f) 正确。
- 10.3 a) 创建对象时应包含关键字 new。
- b) Label 是关键字, 不能作为标识符。
- c) 构造函数 TextField 的两个参数应该互换。
- d) 不存在 RadioGroup 类, 应使用 CheckboxGroup 类。
- e) 布局管理器使用的是 GridLayout, 但添加组件时使用的都是 BorderLayout 位置。
- 10.4 a) 星号是一个通配符, 表示要引入的多个类。
- b) 标签是用于显示不可修改的文本的组件; 文本字段则是一个带有边界的组件, 它既可以显示可编辑的文本, 也可以显示不可编辑的文本。文本字段还可以接收用户输入, 而标签不能。

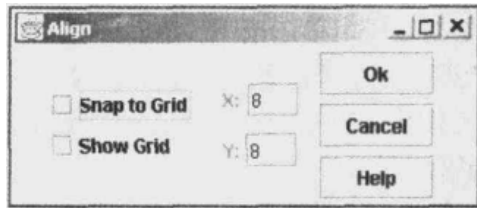
## 练习

10.5 填空:

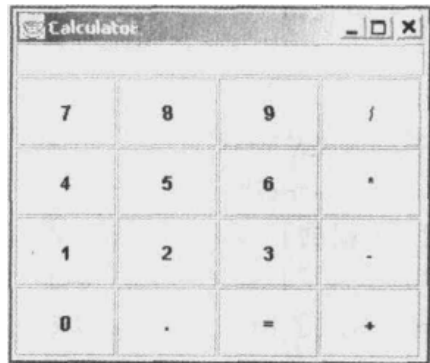
- a) TextField 类是从 \_\_\_\_\_ 类直接继承的。
- b) 本章讨论的 3 种布局管理器是 \_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- c) \_\_\_\_\_ 方法用于将 GUI 组件添加到面板或 applet 上。
- d) 当释放鼠标按键时, 将产生 \_\_\_\_\_ 事件。



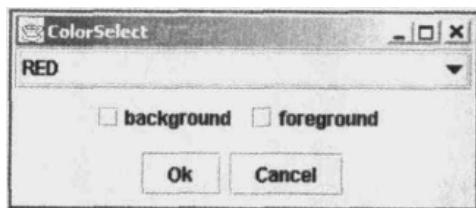
- e) \_\_\_\_\_ 类用于创建一组单选按钮。
- 10.6 判断下列句子是否正确。如果不正确, 请说明原因。
- a) 每个 applet 上只能使用一种布局管理器 ( 添加到 applet 上的每个面板都必须使用和 applet 一样的布局管理器 )。
  - b) 使用 BorderLayout 布局管理器时, GUI 组件可以按任何顺序添加到面板上。
  - c) addItem 方法用于向选择按钮添加选项。
  - d) 单选按钮是一系列互斥的按钮 ( 每次只能有一个为 true )。
  - e) Graphics 对象的 setFont 方法用于设置文本字段的字体。
  - f) 列表对象总包含滚动条。
  - g) Event 实例 target 的数据类型总是 Object。
  - h) Mouse 对象包含一个称为 mouseDrag 的方法。
  - i) BorderLayout 布局管理器提供了 LayoutManager 接口。
  - j) GridLayout 布局管理器提供了 LayoutManager 接口。
  - k) FlowLayout 布局管理器提供了 LayoutManager 接口。
- 10.7 判断下列句子是否正确。如果不正确, 请说明原因。
- a) applet 是一个容器。
  - b) 面板是一个组件。
  - c) 标签是一个容器。
  - d) 列表是一个面板。
  - e) 按钮是一个组件。
  - f) 文本字段是一个对象。
  - g) LayoutManager 是从 Object 继承而来的。
  - h) CheckboxGroup 是从 Component 继承而来的。
- 10.8 找出下列语句中的错误, 并进行改正。
- a) `import java.awt.*// include AWT package`
  - b) `panelObject.GridLayout(8,8);// set GridLayout`
  - c) `button94.add( "Click here!");// add button label`
  - d) `setLayout( new FlowLayout ( FlowLayout.DEFAULT));`
  - e) `if (o.target instanceof Choice)// o is of type Object`
  - f) `add( east, eastButton);// BorderLayout`
  - g) `CheckboxGroup c = new CheckboxGroup( );`  
    `add(c.Checkbox( "Gold", false));`  
    `add(c.Checkbox( "Silver", true));`
  - h) `public boolean mouseMove( Event e, int x, int y)`  
    {  
        `ShowStatus( "mouseMove");`  
    }
- 10.9 创建下面的 GUI ( 注意, 不必为各个组件提供功能 ):



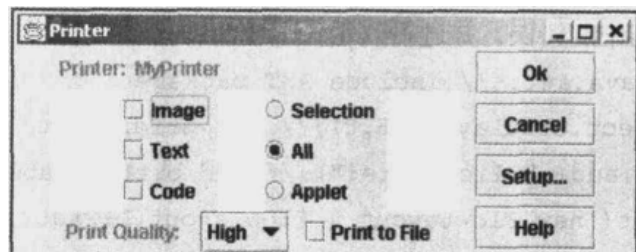
10.10 创建下面的 GUI (注意, 不必为各个组件提供功能):



10.11 创建下面的 GUI (注意, 不必为各个组件提供功能):



10.12 创建下面的 GUI (注意, 不必为各个组件提供功能):



10.13 创建图 10.1 的 GUI。最左边的组件是文本区域, 我们将在下一章讨论文本区域。这里可以使用 `TextArea` 类创建文本区域, 并调用下面的文本区域构造函数:

```
TextArea a=new TextArea ( " Notepad " , 5 , 12 );
```

注意, 不必为 GUI 提供任何功能。

10.14 编写一个将华氏温度转换成摄氏温度的温度转换程序。应从键盘输入华氏温度(通过文本字段输入), 然后再通过文本字段显示转换后的摄氏温度。使用下面的公式进行温度转换:

$$\text{摄氏温度} = 5/9 \times (\text{华氏温度} - 32)$$

10.15 增加练习 10.14 中的温度转换程序的功能, 加入绝对温度的转换。该程序还应允许用户在任何两种温度之间进行转换, 程序中可使用下面的公式(包括练习 10.14 的公式):

绝对温度 = 摄氏温度 + 273

- 10.16 编写一个程序,使用户能够使用鼠标在 applet 中绘制一个矩形。按住鼠标按键,确定矩形的左上角坐标,然后拖动鼠标,在需要的位置上(即矩形的右下角坐标)释放鼠标按键。另外,在状态栏中还要显示矩形面积,利用下面的公式进行计算:

面积 = 宽 × 高

- 10.17 修改练习 10.16 中的程序,使其能画出各种不同的图形,包括椭圆、弧线、直线、圆角矩形及预定义的多边形,还要在状态栏中显示鼠标的当前坐标。
- 10.18 编写一个程序,使用户能够使用鼠标在 applet 上绘制一个图形: c 表示圆形, o 表示椭圆, r 表示矩形, l 表示直线。图形的大小和位置由 mouseDrag 事件确定,在状态栏中显示当前的图形,初始图形默认为圆形。
- 10.19 创建一个可以画图的 applet。用户应能选择所画图形的形状、颜色以及是否在图形上填充颜色。使用本章介绍的一些 GUI 组件来完成这些选择功能,比如选择按钮、单选按钮及复选框。该程序应重写 update 方法,这样用户就可以画出多种图形,并能在 applet 上同时看见所有的图形。该程序应提供一个 Button 对象用于清除 applet 中的图形。为了防止产生混淆,可以在用户拖动鼠标画图时提供一个“橡皮条”效果(即画图时应能看见图形的外部轮廓线)。注意,如果需要这种功能,必须删除重写的 update 方法。在第 14 章中将介绍一些技术,使得程序能够兼有这两种功能。
- 10.20 编写一个程序,使用 System.out.println 语句打印出事件名。提供一个至少包含 4 个或 5 个不同事件的选择按钮,使用户能从选择按钮中选择一个事件进行监视。当发生该事件时,就在状态栏中显示有关该事件的信息。在程序中使用事件的 toString 方法。
- 10.21 编写一个程序,画出一个正方形。当鼠标在绘图区域上移动时,使正方形的左上角随着鼠标指针“移动”,根据鼠标指针的移动路径重画正方形。
- 10.22 修改图 10.27 的程序,加入颜色。在 applet 的底部提供一个“工具栏”,列出下列 6 种颜色: 红色、黑色、深红色、蓝色、绿色和黄色。该工具栏上应包括 6 个按钮,每个按钮上显示相应颜色的名称。当点击某个按钮时,就使用相应的颜色画图,并在状态栏中显示当前的绘图颜色。
- 10.23 修改图 10.27 的程序,允许用户进行擦除。通过背景颜色重画来完成擦除动作,当按住 Shift 键时,就是使用背景颜色画图。
- 10.24 编写一个程序来玩“猜数字”游戏: 首先,程序在 1 到 1 000 之间随机选择一个数字,然后在一个标签上显示下面的信息:

```
I have a number between 1 and 1000 can you guess my number?  
Please enter your first guess.
```

用户可以向一个文本字段中输入所猜的数字。每当用户输入一个答案时,背景颜色就改成红色或蓝色,红色表示所猜数字“偏大”,蓝色表示所猜数字“偏小”。另外,再给出一个不可编辑的文本字段,显示“Too High”或“Too Low”信息,以帮助用户调整到正确的答案。当用户猜到正确的数字时,该文本字段将显示信息“Correct !”,同时,使得用于输入的文本字段变成非编辑状态。另外,提供一个重新开始玩游戏的按钮,当点击该按钮时,将产生一个新的随机数字,同时第一个文本字段又变成可编辑的。

- 10.25 将程序执行过程中产生的事件显示出来,有助于理解这些事件是何时产生的以及如何产生的。编写一个 applet,使其能够产生并处理本章中讨论过的各个事件。在该 applet 中重写 `action`、`mouseUp`、`mouseDown`、`mouseMove`、`mouseDrag`、`mouseEnter`、`mouseExit`、`keyUp` 和 `keyDown` 方法,当事件发生时显示相应的信息。利用 `Event` 类的 `toString` 方法将事件转换成一个字符串显示出来,`toString` 方法将创建一个字符串,其中包含了事件对象中的所有信息。
- 10.26 修改练习 10.19 的程序,使用户可以选择字体和字体大小,并可以向一个文本字段中输入文本。当用户按下回车键时,以选中的字体和大小显示该文本。然后,再进一步修改该程序,使用户可以指定文本显示的位置。

## 第11章 图形用户界面组件（二）

### 教学目标

- 扩展图形用户界面的功能
- 学会创建和使用文本区域
- 学会创建和使用画板
- 学会创建和使用滚动条
- 学会创建定制组件
- 学会创建和使用框架
- 学会创建和使用菜单
- 学会创建和使用对话框
- 学会使用高级的布局管理器

### 11.1 简介

本章继续介绍图形用户界面（GUI），这里我们将讨论更高级的组件和布局管理器，以及如何建立比较复杂的 GUI。图 11.1 中列出了本章将要讨论的组件的继承层次结构。

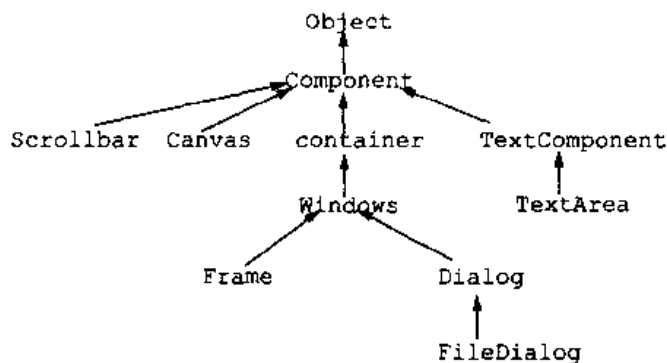


图 11.1 本章将要讨论的组件在 java.awt 中的继承层次结构

### 11.2 文本区域

文本区域是一个可以显示多行文本的区域。与 `TextField` 类一样，`TextArea` 类是从 `TextComponent` 继承而来的，图 11.2 中给出了 `TextArea` 类的两个构造函数。

图 11.3 中的程序说明了图 11.2 中两个构造函数的用法。其中一个文本区域所显示的文本不允许用户进行修改，而另一个文本区域则允许用户输入信息。

**TextArea 类的构造函数**

```

public TextArea (
    int rows ,           //number of visible rows
    int columns )       //number of visible columns
    创建一个文本区域，它的行数和列数由参数指定

public TextArea (
    String s,
    int rows,
    int columns )
    创建一个包含字符串 s 的文本区域，其行数和列数由参数指定

```

图 11.2 TextArea 类的构造函数

```

1  // Fig. 11.3: MyTextArea.java
2  // Creating TextArea objects.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class MyTextArea extends Applet {
7      private TextArea t1, t2;
8
9      public void init()
10     {
11         t1 = new TextArea( "This text is read-only!",
12                             10, 20 );    // 10 x 20
13         t2 = new TextArea( 10, 20 );
14
15         t1.setEditable( false );        // read-only
16
17         // set layout
18         setLayout( new FlowLayout( FlowLayout.LEFT ) );
19
20         add( t1 );
21         add( t2 );
22     }
23 }

```

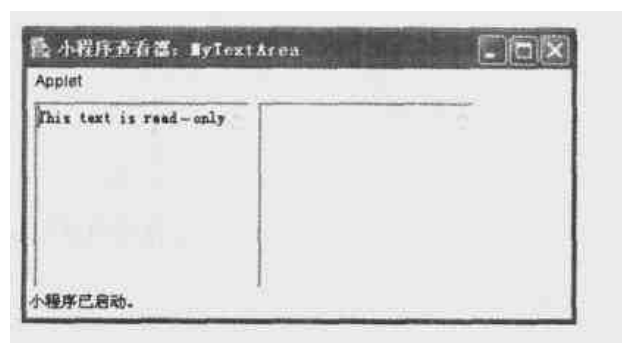


图 11.3 创建 TextArea 对象

程序中使用 TextArea 类创建文本区域 t1 和 t2。下列语句：

```
t1 = new TextArea ( " This text is read-only ! " , 10 , 20 );
```

用于创建一个 10 行、20 列的文本区域 t1，并显示文本 “This text is read-only !” 下列语句：

```
t2 = new TextArea ( 10,20 ) ;
```

用于创建文本区域 t2。文本区域 t2 包括 10 行、20 列，初始时未显示任何文本。

使用 `setEditable` 方法将文本区域 t1 设置为不可编辑的，并将文本区域 t2 设置为可编辑的，即用户可以修改。该 applet 的布局管理器将设置成左对齐的 `FlowLayout`，使用 `add` 方法分别将每个文本区域添加到 applet 上。

文本区域总是带有一个垂直滚动条和一个水平滚动条，仅当文本区域中的文本超出文本区域的可见范围时，滚动条才能滚动。程序员无法直接控制这两个滚动条，我们将在 11.4 节讨论可控制的滚动条组件。

在文本字段中，按下回车键会产生一个 `action` 事件。文本区域与此不同，它不产生任何可由 `handleEvent` 方法或 `action` 方法处理的事件。要从文本区域中获取文本，则需要一个外部事件来触发——比如单击一个按钮。这个过程类似于从一个列表中获取多个选项（请参见第 10 章）。

图 11.4 中程序的功能是将一个文本区域中选中的文本复制到另一个文本区域中，当点击按钮时，就执行复制文本的动作。

文本区域对象 t1 和 t2 均已实例化，每个文本区域都是 5 行、20 列。下列语句：

```
t1.setText ( s );
```

使用 `TextComponent` 类的 `setText` 方法将字符串 s 设置到 t1 中。实例化按钮对象 b，并设置了标签“Copy>>>”。该 applet 的布局管理器设置成左对齐的 `FlowLayout`，各组件之间的水平距离为 5 个像素，垂直距离也为 5 个像素。这里使用 `add` 方法将各个组件添加到 applet 中。

---

```

1 // Fig. 11.4: MyTextArea2.java
2 // Copying selected text from one text area to another.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyTextArea2 extends Applet {
7     private TextArea t1, t2;
8     private Button b;
9
10    public void init()
11    {
12        String s = "The big bad boy stepped up \n" +
13                  "to the microphone. The crowd \n" +
14                  "sat on the edge of their seats \n" +
15                  "and waited...";
16
17        t1 = new TextArea( 5, 20 );
18        t1.setText( s ); // add text to t1
19
20        t2 = new TextArea( 5, 20 );
21        b = new Button( "Copy >>>" );
22
23        // set layout
24        setLayout( new FlowLayout( FlowLayout.LEFT, 5, 5 ) );
25
26        add( t1 );
27        add( b );
28        add( t2 );
29    }

```

```

30
31     public boolean action( Event e, Object o )
32     {
33         if ( e.target == b ) {
34             t2.setText( t1.getSelectedText() );
35             return true;
36         }
37
38         return false;
39     }
40 }

```

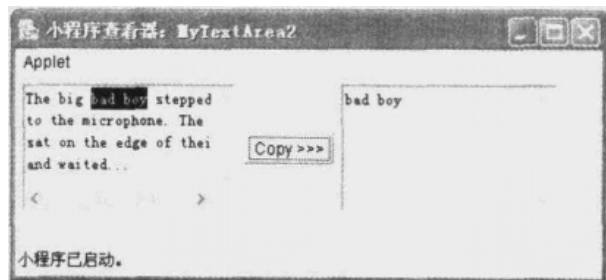


图 11.4 将选中的文本复制到另一个文本区域中

该程序重写了 `action` 方法。当点击 “Copy>>>” 按钮时，`t1` 中所有选中的文本都将放置到 `t2` 中。下面的语句用于完成复制操作：

```
t2.setText ( t1.getSelectedText ( ) );
```

这里使用 `TextComponent` 类的 `getSelectedText` 方法返回文本区域中选中的文本字符串，然后再使用 `setText` 方法将该文本字符串放置到 `t2` 中。如果在点击按钮时，`t1` 中没有选中的文本字符串，那么就不会产生复制动作。可以通过鼠标高亮显示选中的文本。

### 11.3 画板

画板 (canvas) 是一个可以画图 and 接收鼠标事件的组件，`Canvas` 类是从 `Component` 类继承的，画板组件有它自己的图形环境。

#### 软件工程视点 11.1

直接绘制在 applet 上的图形会被其他组件覆盖，而画板提供了一个专用的绘图区域，从而解决了这个问题。

图 11.5 的程序创建了一个最简单的画板，`Component` 类的 `setBackground` 方法用于将画板的背景改成绿色。不过，该程序只是创建了一个绿色画板，没有进行其他操作。

下面的语句是调用 `Canvas` 构造函数，并创建一个画板对象 `c`：

```
c = newCanvas ( );
```

可以使用 `setBackground` 方法来修改画板 `c` 的背景颜色。画板初始创建时没有设置大小，下面的语句是使用 `Component` 类的 `resize` 方法重新设置画板的宽度和高度：

```
c.resize( 250 , 60 );
```

可以使用 `add` 方法将该画板对象添加到 applet 上。



## 常见编程错误 11.1

如果使用画板时忘记调用 `resize` 方法, 则会产生运行时的逻辑错误, 这时画板将不可见。

```

1  // Fig. 11.5: CreateCanvas.java
2  // Creating a Canvas.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  public class CreateCanvas extends Applet {
7      private Canvas c;
8
9      public void init()
10     {
11         c = new Canvas(); // instantiate object
12         c.setBackground( Color.green );
13         c.resize( 250, 60 );
14
15         add( c );
16     }
17 }

```

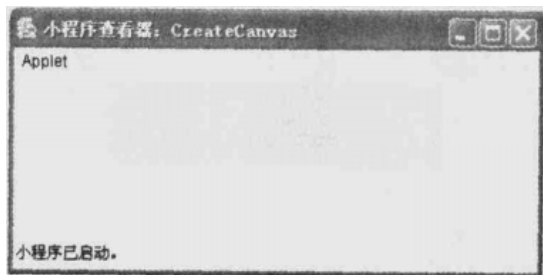


图 11.5 创建一个画板

`NewCanvas` 类包含一个实例变量 `shape`, 表示当前所画图形的形状。`NewCanvas` 类的 `paint` 方法用于在画板上画图, 如果 `shape` 等于 1, 那么就调用 `fillOval` 方法绘制一个填充圆形; 否则, 就调用 `fillRect` 方法绘制一个填充矩形。在 `setShape` 方法中, 首先设置变量 `shape` 的值, 然后调用 `repaint` 方法来刷新 applet。当调用 `paint` 时, 相应的图形就在画板上显示出来。

在图 11.5 中创建的画板没有绘图功能, 因为该画板的 `paint` 方法 (从 `Component` 类继承) 未执行任何操作。要想使 `paint` 方法能够在画板上画图, 必须将其重写, 这可以通过创建一个 `Canvas` 的子类来完成。在画板上画图的坐标从画板的左上角开始, 即左上角的坐标为 (0,0)。

图 11.6 的程序创建了一个 `NewCanvas` 类, 它是 `Canvas` 类的扩展。该程序将根据用户所按下的按钮来确定是画一个填充圆形还是画一个填充正方形。

```

1  // Fig. 11.6: MyCanvas.java
2  // Extending the Canvas class.
3  import java.applet.Applet;
4  import java.awt.*;
5
6  class NewCanvas extends Canvas {
7      private int shape;
8
9      public void paint( Graphics g )
10     {
11         if ( shape == 1 )

```

```

12         g.fillOval( 50, 10, 60, 60 );
13     else // shape == 2
14         g.fillRect( 50, 10, 60, 60 );
15     }
16
17     public void setShape( int s )
18     {
19         shape = s;
20         repaint();
21     }
22 }
23
24 public class MyCanvas extends Applet {
25     private Panel p;
26     private NewCanvas c;
27     private Button circleButton, squareButton;
28
29     public void init()
30     {
31         p = new Panel();
32         c = new NewCanvas(); // instantiate canvas
33
34         c.resize( 185, 125 ); // resize canvas
35
36         squareButton = new Button( "Square" );
37         circleButton = new Button( "Circle" );
38
39         p.setLayout( new BorderLayout() );
40         p.add( "North", circleButton );
41         p.add( "South", squareButton );
42
43         setLayout( new BorderLayout() );
44         add( "West", p );
45         add( "East", c ); // add canvas
46     }
47
48     public boolean action( Event e, Object o )
49     {
50         if ( e.target == circleButton )
51             c.setShape( 1 );
52         else // squareButton
53             c.setShape( 2 );
54
55         return true;
56     }
57 }

```

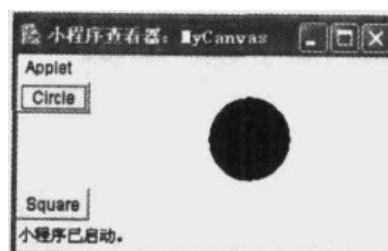
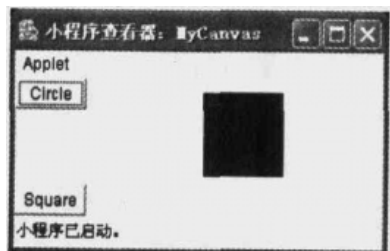


图 11.6 扩展的 Canvas 类

在 `MyCanvas` 类中声明了一个 `Panel` 对象、一个 `NewCanvas` 对象及两个 `Button` 对象, 通过调用构造函数来创建 `Panel` 对象 `p`。下面的语句调用了 `NewCanvas` 的构造函数:

```
c = new NewCanvas ( );
```

画板的大小设置成 185 像素 × 125 像素。通过调用 `Button` 构造函数来创建按钮对象 `squareButton` 和 `circleButton`, 该面板的布局管理器设置成 `BorderLayout`, 这两个按钮对象都将添加到面板上。`applet` 的布局管理器也设置成 `BorderLayout`, 前面创建的面板和画板都添加到该 `applet` 上。`action` 方法将被重写, 如果按下 `circleButton` 按钮, 那么调用 `setShape` 方法时参数为 1, 否则参数为 2, 然后就利用相应的图形更新 `NewCanvas`。

画板不产生任何事件, 但是能够接收鼠标事件。图 11.7 中程序的功能是使用鼠标在画板上绘制椭圆。

其中 `NewCanvas` 类是 `Canvas` 类的扩展, 变量 `width` 和 `height` 中分别保存着椭圆的宽度和高度, 另外两个变量中保存的是鼠标指针的 `x` 坐标和 `y` 坐标。

---

```
1 // Fig. 11.7: NewCanvas.java
2 // Capturing mouse events with a Canvas.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 class NewCanvas extends Canvas {
7     private int width, height;
8     private int x1, y1;
9
10    public void paint( Graphics g )
11    {
12        g.drawOval( x1, y1, width, height );
13    }
14
15    public boolean mouseDown( Event e, int x, int y )
16    {
17        x1 = x;
18        y1 = y;
19        return true;
20    }
21
22    public boolean mouseUp( Event e, int x, int y )
23    {
24        width = Math.abs( x - x1 );
25        height = Math.abs( y - y1 );
26
27        // determine the upper left point of
28        // the bounding rectangle
29        x1 = Math.min( x1, x );
30        y1 = Math.min( y1, y );
31
32        repaint();
33        return true;
34    }
35 }
36
37 public class MyCanvas extends Applet {
```

```

38     private NewCanvas c;
39
40     public void init()
41     {
42         c = new NewCanvas();
43         c.resize( 185, 70 ); // resize canvas
44         c.setBackground( Color.yellow );
45
46         add( c ); // add canvas to applet
47     }
48 }

```

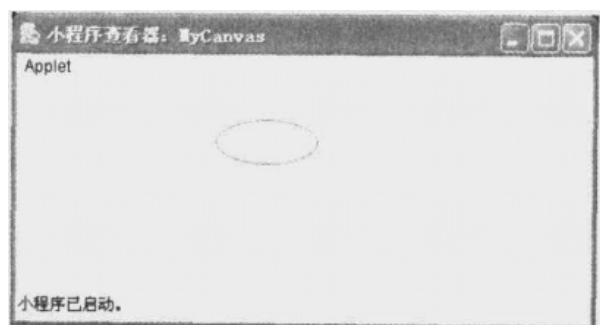


图 11.7 在画板上接收鼠标事件

在这个类中重写了 `paint` 方法，它调用 `drawOval` 方法绘制椭圆，指定了包含椭圆的矩形的宽度、高度，以及左上角坐标(`xDown`, `yDown`)。绘图将严格限制在画板内，任何画到画板外面的图形都将剪切掉。

本例中重写了 `mouseDown` 方法和 `mouseUp` 方法。在 `mouseDown` 方法中设置了实例变量 `x1` 和 `y1` 的值，在 `mouseUp` 方法中通过下面的语句设置了 `width` 和 `height` 的值：

```

width = Math.abs ( x - x1 );
height = Math.abs ( y - y1 );

```

利用产生 `mouseUp` 事件时的 `x` 值减去 `x1`，并将 `y` 值减去 `y1`。椭圆的宽度和高度必须是正数，否则就无法显示出来，所以使用 `Math` 类的 `abs` 方法求出结果的绝对值。包含椭圆的矩形的左上角坐标应是两对坐标中较小的一个，所以使用 `Math` 类的 `min` 方法求出最小值，并赋给 `x1` 和 `y1`。然后调用 `repaint` 方法重画画板。

`MyCanvas` 类创建了一个 `NewCanvas` 对象 `c`。该画板重新设置了大小，将背景颜色设置成黄色，然后添加到 `applet` 中。

## 11.4 滚动条

滚动条 (scrollbar) 是一个可以在某个整数范围内“滚动”的组件。可以使用 `Scrollbar` 类创建独立的滚动条，`Scrollbar` 类是从 `Component` 类继承而来的。图 11.8 中显示了一个独立的水平滚动条。

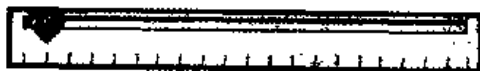


图 11.8 一个滚动条组件

滚动条可以是水平方向或垂直方向。水平滚动条的滚动箭头为左右方向,左端表示最小值,右端表示最大值。垂直滚动条的滚动箭头为上下方向,顶端表示最小值,底端表示最大值。滚动块(也称为滑块)的位置指示了滚动条的当前值。

Scrollbar类有3个构造函数,图11.9中给出了其中的一个。滚动条的方向由两个静态常量确定,在图11.9中列出了这些常量和Scrollbar构造函数。

| 滚动条常量和 Scrollbar 构造函数                              |                                           |
|----------------------------------------------------|-------------------------------------------|
| <code>public final static int HORIZONTAL</code>    | 表示滚动条为水平方向的常量                             |
| <code>public final static int VERTICAL</code>      | 表示滚动条为垂直方向的常量                             |
| <code>public Scrollbar (</code>                    |                                           |
| <code>int orientation,</code>                      | <code>//HORIZONTAL or VERTICAL</code>     |
| <code>int initialValue,</code>                     | <code>//initial value of scrollbar</code> |
| <code>int visibleArea,</code>                      | <code>//scroll box size</code>            |
| <code>int minimumValue,</code>                     | <code>//minimum value of scrollbar</code> |
| <code>int maximumValue )</code>                    | <code>//maximum value of scrollbar</code> |
| 创建一个滚动条,分别使用不同的参数指定其方向(水平或垂直)、初始值、滚动块的位移大小、最小值和最大值 |                                           |

图 11.9 滚动条常量和 Scrollbar 构造函数

上述构造函数含有5个整形参数。第一个参数的值为Scrollbar.HORIZONTAL或Scrollbar.VERTICAL,用于指定滚动条的方向。第二个参数用于指定滚动条的初始值,开始时滚动块就位于所指定的位置。第三个参数用于指定滚动块的位移大小,当用户在滚动条的滚动箭头和滚动块之间点击时,将由这个值来确定滚动块应该移动的距离。第四个参数用于指定滚动条的最小值,如果初始值小于最小值,那么初始值将等于这个最小值。第五个参数用于指定滚动条的最大值,如果初始值大于最大值,那么初始值将等于这个最大值。

#### 常见编程错误 11.2

调用图11.9的Scrollbar构造函数时,如果第一个参数既不是Scrollbar.HORIZONTAL也不是Scrollbar.VERTICAL,那么会引发IllegalArgumentExpection异常。

图11.10中程序的功能是改变画板上所画椭圆的大小,这一功能是通过一个水平滚动条和一个垂直滚动条来完成的。

本例中创建了两个滚动条upDown和leftRight。下面的语句用于创建upDown:

```
upDown = new Scrollbar( Scrollbar.VERTICAL, 100, 0 , 0 , 200 );
```

滚动条upDown的方向为垂直的,初始值为100,滚动块的位移大小设置成0,滚动条的最小值是0,最大值是200。水平滚动条leftRight的创建方法与upDown类似。

```

1    // Fig. 11.10: Scroll.java
2    // Using Scrollbars to size an oval.
3    import java.applet.Applet;
4    import java.awt.*;
5
6    public class Scroll extends Applet {
7        private Scrollbar upDown, leftRight;
8        private MyCanvas theCanvas;
```

```
9      private Panel thePanel;
10
11      public void init()
12      {
13          // create scrollbars
14          upDown = new Scrollbar( Scrollbar.VERTICAL, 100,
15                                0, 0, 200 );
16
17          leftRight = new Scrollbar( Scrollbar.HORIZONTAL, 100,
18                                    0, 0, 200 );
19
20          // create canvas
21          theCanvas = new MyCanvas();
22          theCanvas.resize( 200, 200 );
23          theCanvas.setBackground( Color.yellow );
24
25          thePanel = new Panel(); // create panel
26
27          // set layouts
28          setLayout( new BorderLayout() );
29          thePanel.setLayout( new BorderLayout() );
30
31          // layout components
32          add( "West", upDown );
33          thePanel.add( "South", leftRight );
34          thePanel.add( "Center", theCanvas );
35          add( "Center", thePanel );
36      }
37
38      public boolean handleEvent( Event e )
39      {
40          if ( e.target instanceof Scrollbar ) {
41
42              if ( e.target == upDown )
43                  theCanvas.setHeight( upDown.getValue() );
44              else // leftRight
45                  theCanvas.setWidth( leftRight.getValue() );
46
47              showStatus( "e.arg is " + String.valueOf( e.arg ) );
48              theCanvas.repaint(); // repaint canvas
49              return true;
50          }
51
52          // not one of our scroll bars
53          // do the default handling ( base class )
54          return super.handleEvent( e );
55      }
56  }
57
58  class MyCanvas extends Canvas {
59      private int width, height;
60
61      public MyCanvas()
62      {
63          setWidth( 100 );
64          setHeight( 100 );
65      }
```

```

66
67     public void setWidth( int w )
68     {     width = w;     }
69
70     public void setHeight( int h )
71     {     height = h;     }
72
73     public void paint( Graphics g )
74     {
75         g.setColor( Color.black );
76         g.drawOval( 0, 0, width, height );
77     }
78 }

```

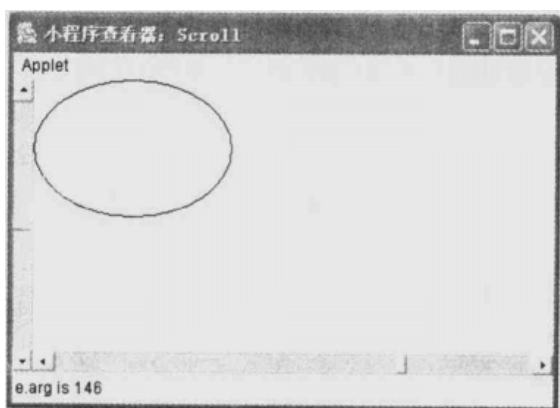


图 11.10 使用滚动条来改变椭圆的大小

Canvas 类的扩展类 MyCanvas 中包含一个构造函数、一个 setWidth 方法、一个 setHeight 方法和一个 paint 方法。setWidth 和 setHeight 方法分别用于设置私有变量 width 和 height 的值, paint 方法用于将当前的颜色设置成黑色, 并绘制一个椭圆。当创建 MyCanvas 类的 theCanvas 对象时, 改变它的大小, 并将背景颜色设置成黄色。

这里我们创建了面板对象 thePanel, 将 applet 和面板 thePanel 的布局管理器设置为 BorderLayout。将滚动条 upDown 放置到 applet 的西区, 将滚动条 leftRight 放置到 thePanel 的南区, 将画板 theCanvas 放置到 thePanel 的中区。然后, 将整个面板 thePanel 放置到 applet 的中区。

在 Scroll 类中, 本例重写了 handleEvent 方法。当使用鼠标点击滚动箭头、移动滚动块或点击滚动箭头和滚动块之间的区域时, 将会产生滚动条事件, 使用 handleEvent 方法处理滚动条事件。下面的语句用于判断产生事件的组件是否为滚动条:

```
if( e.target instanceof Scrollbar )
```

如果是, 那么再通过接下来的 if/else 语句判断该事件是否为滚动条事件。如果是 upDown 产生的事件, 就执行下面的语句:

```
theCanvas.setHeight ( upDown.getValue ( ) );
```

setHeight 方法传入的是滚动条 upDown 的当前值, getValue 方法的功能是返回滚动条的当前值。如果是 leftRight 产生的事件, 就执行下面的语句:

```
theCanvas.setWidth ( leftRight.getValue ( ) );
```

setWidth 方法传入的是滚动条 leftRight 的当前值。

对于一个滚动条来说, `Event` 的实例变量 `arg` 是一个包含滚动条的值的整型对象。`showStatus` 方法将在状态栏中显示 `e.arg` 的值, `repaint` 方法将刷新 applet 上的组件。

如果滚动条产生了事件, 那么 `handleEvent` 方法将返回 `true` 值, 否则就调用超类 `handleEvent` 进行默认的事件处理, 将 `Event` 传递给原始的 `handleEvent` 方法(从 `Scroll` 类中继承)。原始的 `handleEvent` 方法的功能是接收一个事件, 确定该事件的类型, 然后将该事件传递给另一个事件处理方法, 比如 `action`、`mouseUp`、`keyDown` 等。

## 11.5 定制组件

Java 允许程序员创建自己的定制组件, 定制组件是由程序员创建而非系统提供的组件。

一般情况下, 定制组件是通过扩展 `Canvas` 类或 `Panel` 类来创建的。不能使用 `Component` 类, 因为它没有公有构造函数。如果定制组件不需要 `Container` 类的方法, 那么可以扩展 `Canvas` 类。

如果定制组件中包含其他的组件, 那么应当扩展 `Panel` 类。为了添加多个组件, 需要使用 `Container` 类的方法。包含多个组件的定制组件称为组合组件。在这一节里, 我们将主要讨论组合组件。

### 软件工程视点 11.2

创建定制组件的功能是 Java 的扩展功能之一, 也是 Java 最具吸引力的特性之一。

### 编程技巧 11.1

如果在创建定制组件时不需要 `Container` 方法, 那么就扩展 `Canvas` 类而不是 `Panel` 类。

图 11.11 的程序创建了一个组合组件——带有标签的文本字段。这里扩展了 `Panel` 类, 用于创建定制组件类 `LabelTextField`, 该组件包含一个 `Label` 组件和一个 `TextField` 组件。它首先将文本放到文本字段中, 并在标签中显示出来。当点击按钮时, 在状态栏中显示文本字段中的文本。

```
1 // Fig. 11.11: Test.java
2 // Creating a custom component.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class Test extends Applet {
7     private LabelTextField customComponent;
8     private Button button1;
9
10    public void init()
11    {
12        customComponent = new LabelTextField( "label text" );
13        button1 = new Button( "Button" );
14
15        add( button1 );
16        add( customComponent );
17    }
18
19    // button event is handled here
20    public boolean action( Event e, Object o )
21    {
22        String s = "Textfield text: ";
23        showStatus( s + customComponent.getFieldText() );
24        return true;
25    }
26 }
```



```

25     }
26 }
27
28 class LabelTextField extends Panel {
29     private Label theLabel;
30     private TextField theTextField;
31
32     public LabelTextField( String s )
33     {
34         setLayout( new BorderLayout() );
35         theLabel = new Label( s );
36         theTextField = new TextField( 10 );
37
38         add( "North", theLabel );
39         add( "South", theTextField );
40     }
41
42     // handles its own textfield event
43     public boolean action( Event e, Object o )
44     {
45         setLabelText( e.arg.toString() );
46         return true;
47     }
48
49     // setLabelText is programmer-defined
50     public void setLabelText( String s )
51     {
52         theLabel.setText( s );
53     }
54
55     // getFieldText is programmer-defined
56     public String getFieldText()
57     {
58         return theTextField.getText();
59     }
60 }

```

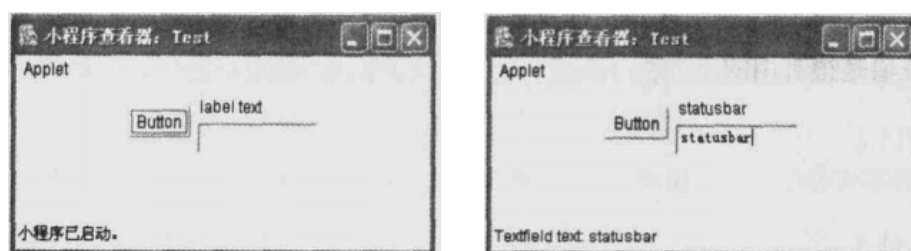


图 11.11 创建定制组件

在Test类中,创建了一个LabelTextField类的对象customComponent和一个按钮对象button1。下面的语句用于创建LabelTextField对象:

```
customComponent = new LabelTextField ( "label text, " );
```

LabelTextField构造函数的参数是字符串“label text”,按钮对象button1是通过调用Button构造函数来创建的,然后使用add方法将button1和customComponent添加到applet中。

程序中重写了 `action` 方法。当点击按钮时，在 `action` 方法中处理产生的事件。下面的语句用于在状态栏中显示字符串 `s`，该字符串是通过 `LabelTextField` 的 `getFieldText` 方法获取的：

```
string s = "Textfield text:";
showStatus ( s + v.getFieldText ( ) );
```

这个 applet 的 `action` 方法没有处理文本字段所生成的事件，这些事件由 `LabelTextField` 类的 `action` 方法处理。

`LabelTextField` 类是 `Panel` 类的扩展，因此 `LabelTextField` 是一种容器。`LabelTextField` 有一个 `Label` 和一个 `TextField` 实例变量。

`LabelTextField` 构造函数带有一个字符串参数，用于在标签中显示。该容器的布局管理器设置为 `BorderLayout`，并创建了实例变量 `theLabel` 和 `theTextField`，对象 `theTextField` 的宽度是 10 列。将标签放置到容器的北区，而文本字段则放置到南区。

`LabelTextField` 的 `setLabelText` 方法用于设置标签组件的文本。下面的语句中使用了 `Label` 类的 `setText` 方法来设置标签的文本：

```
theLabel.setText ( s );
```

注意，标签对象 `theLabel` 是私有的，因此本类之外的对象是不能访问它的。`setLabelText` 提供了一个设置标签文本的公有接口。

`LabelTextField` 还提供了一个 `getFieldText` 方法，用于返回 `private` 文本字段中的文本。下面的语句使用 `TextComponent` 的 `getText` 方法，返回文本字段中的文本：

```
return theTextField.getText ( ) ;
```

`LabelTextField` 类通过重写 `action` 方法来处理它自己的活动事件。下面的语句中使用了 `setLabelText` 方法来设置标签文本，文本的内容是 `e.arg` 的值：

```
setLabelText ( e.arg.toString ( ) );
```

`action` 方法返回 `true` 值，表示事件已成功处理。

如果 `action` 方法返回 `false` 值，那么就将该事件送到 applet 容器中。applet 的 `handleEvent` 方法将该事件送到 applet 的 `action` 方法中。如果要求定制组件部分地或者有选择地处理某个组件事件，那么返回 `false` 值是很有用的。

#### 编程技巧 11.2

当定制组件不需要 `Container` 类的方法时，就使用 `Canvas` 类的扩展类，而不是 `Panel` 类。

#### 编程技巧 11.3

在定制组件中，应提供 `get` 和 `set` 方法来访问私有数据。

## 11.6 框架

框架是一个带有标题栏和边界的窗口，由 `Window` 类的扩展类 `Frame` 创建，`Window` 类是从 `Container` 类继承而来的。`Window` 类中包含一些用于窗口管理的方法。`Window` 对象没有标题栏或边界。窗口和框架的默认布局管理器是 `BorderLayout`。

**常见编程错误 11.3**

窗口和框架的默认布局管理器是 BorderLayout。如果在向窗口和框架添加组件时忘记指定 “North”、“South”、“East”、“West” 或 “Center”，则会产生运行时的逻辑错误。该组件也显示不出来。

Frame 通常用于建立有窗口的应用程序(不需要浏览器就可运行的程序),偶尔也可用于 applet。在这一节中,我们将讨论使用 Frame 和基于 Frame 的 applet。

图 11.12 的程序创建了一个带有单个按钮的 applet。当点击按钮时,将显示一个带有标签的框架。

```
1 // Fig. 11.12: MyFrame.java
2 // Creating a Frame from an applet.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyFrame extends Applet {
7     private Frame f;
8     private Font x;
9     private String s;
10    private Label l;
11    private Button showFrame;
12
13    public void init()
14    {
15        String s2 = "Click here to see a Frame";
16
17        s = "FRAME";
18        x = new Font( "Courier", Font.BOLD, 36 );
19        showFrame = new Button( s2 );
20        add( showFrame );
21    }
22
23    public boolean action( Event e, Object o )
24    {
25        if ( e.target == showFrame ) {
26
27            if ( f != null ) {
28                f.hide(); // hide frame
29                f.dispose(); // free resources
30            }
31
32            // create frame
33            f = new Frame( "A Frame!" );
34
35            // create label
36            l = new Label( s );
37            l.setFont( x );
38
39            // change layout
40            f.setLayout( new FlowLayout() );
41
42            // add label to frame
43            f.add( l );
44
45            // size and show frame
```

```

46         f.resize( 300, 100 );
47         f.show();    // display frame
48     }
49
50     return true;
51 }
52 }

```

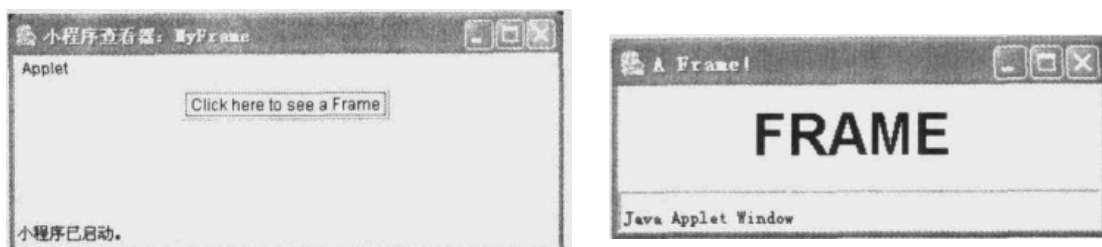


图 11.12 在 applet 上创建一个框架

MyFrame 类创建了一个 Frame 对象、一个 Font 对象、一个 String 对象、一个 Label 对象和一个 Button 对象。

字符串对象 s2 和 s 分别利用下面两条语句进行赋值：

```

Strings2 = " Click here to see a Frame ";
s = " Frame ";

```

本例创建了一个 36 点阵的黑体 Courier 字体。在创建按钮 showFrame 的构造函数中使用了参数 s2 作为该按钮的标签。该按钮通过 add 方法添加到 applet 上。

本例中重写了 action 方法。首先通过一个 if 语句，判断 showFrame 按钮是否产生了事件。如果是，那么再使用下面的条件来判断框架对象 f 是否不为空：

```
f!=null
```

如果不为空，就执行下面的语句：

```

f.hide ( );
f.dispose ( );

```

Component 的 hide 方法（从 Frame 类中继承）将框架从屏幕上隐藏起来，这样框架就不再可见了。然后，再通过 Frame 的 dispose 方法，将窗口系统分配给该框架的资源释放。在 action 方法中提供这两条语句，主要是因为在一个程序中只能存在一个框架。

#### 编程技巧 11.4

在调用 dispose 方法之前应先将框架隐藏起来。有时，由于框架的位图图像仍然在屏幕上显示，因此无法释放它所占用的资源。

#### 性能提示 11.1

释放一个框架就是释放窗口系统的资源。

下面的语句通过调用 Frame 构造函数来创建一个 Frame 对象：

```
f = new Frame ( " A Frame ! " );
```

字符串“A Frame!”是显示在框架的标题栏（位于窗口顶部）中的文本。

标签对象利用 Label 构造函数进行创建,其参数为字符串 s,该字符串将显示在标签上,该标签的字体是通过 setFont 方法设置的。框架的布局管理器将设置成 FlowLayout,然后利用下面的语句将标签添加到框架中:

```
f.add ( l );
```

框架在显示之前必须设置大小。一个框架没有默认的大小值,但其最小应能显示标题栏。Component 的 resize 方法用于设置框架的大小,指定其宽度和高度。下面的语句是将框架显示在屏幕上:

```
f.show ( );
```

Component 的 show 方法用于显示一个组件。默认时框架是不可见的。程序员必须显式地调用 show 方法来显示框架。其他大多数的组件在添加到 applet 或应用程序上时将自动显示出来。

#### 常见的编程错误 11.4

如果在使用框架时忘记调用 resize 方法,则会引发运行时的逻辑错误;通常只有框架的标题栏能显示出来。

#### 常见的编程错误 11.5

如果在使用框架时忘记调用 show 方法,则会引发运行时的逻辑错误,框架也无法显示出来。

大多数窗口的左上角或右上角上都有一个图标,用于关闭窗口和终止程序。如果在 Java 框架上点击该图标(又称退出图标),则不会产生任何效果,除非程序员编写一段代码来关闭框架。本例中,只有 applet 终止(例如退出 applet)时才会关闭框架。我们将在下个例子中编写一段程序,使其在点击退出图标时能关闭框架。

图 11.12 的程序说明了如何创建、显示和删除框架。最后创建出的框架没有任何功能。与 Canvas 类相类似,通常扩展 Frame 类来完成某一功能。图 11.13 的程序创建了一个带有单个按钮的 applet。当点击该按钮时,就会显示一个四周带有 4 个按钮的框架。该框架是 Frame 类的扩展类。当点击框架上的按钮时,框架的背景就变成由该按钮所指定的颜色。

```
1 // Fig. 11.13: MyFrame2.java
2 // Creating a subclass of Frame.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class MyFrame2 extends Applet {
7     private DemoFrame f;
8     private Button showFrame;
9
10    public void init()
11    {
12        String s = "Click here to see the Frame";
13
14        showFrame = new Button( s );
15        add( showFrame );
16    }
17
18    public boolean action( Event e, Object o )
19    {
20        if ( e.target == showFrame ) {
21            String s = "This frame does something!";
```

```
22
23         if ( f != null ) {
24             f.hide();        // hide frame
25             f.dispose();     // free resources
26         }
27
28         f = new DemoFrame( s );    // instantiate frame
29         f.setSize( 300, 200 );    // resize frame
30
31         // do not allow the frame to be resized
32         f.setResizable( false );
33
34         f.show();    // display frame
35     }
36
37     return true;
38 }
39 }
40
41 class DemoFrame extends Frame {
42     private Button a, b, c, d;
43
44     public DemoFrame( String s )
45     {
46         // call base class constructor
47         super( s );
48
49         a = new Button( "yellow" );
50         b = new Button( "red" );
51         c = new Button( "blue" );
52         d = new Button( "green" );
53
54         // default layout is border layout
55         add( "North", a );
56         add( "East", b );
57         add( "South", c );
58         add( "West", d );
59     }
60
61     public boolean handleEvent( Event e )
62     {
63         if ( e.id == Event.WINDOW_DESTROY ) {
64             hide();    // hide frame
65             dispose(); // free resources
66             return true;
67         }
68
69         return super.handleEvent( e );
70     }
71
72     public boolean action( Event e, Object o )
73     {
74         if ( e.target == a )
75             setBackground( Color.yellow );
76         else if ( e.target == b )
77             setBackground( Color.red );
```

```

78         else if ( e.target == c )
79             setBackground( Color.blue );
80         else // d button
81             setBackground( Color.green );
82
83         repaint(); // update color change
84         return true;
85     }
86 }

```

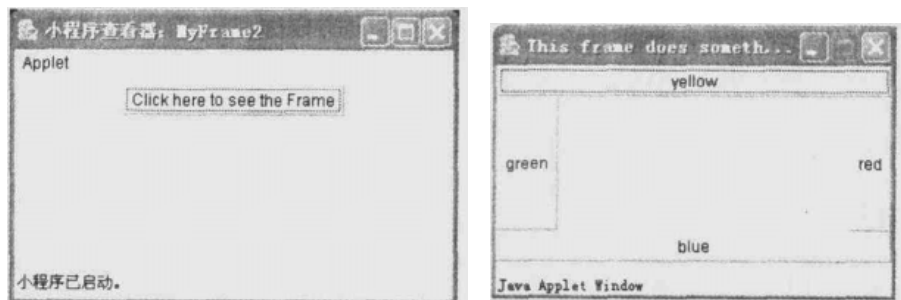


图 11.13 创建一个 Frame 的子类

Applet 的扩展类 MyFrame2 声明了一个 DemoFrame 对象和一个 Button 对象。在 init 方法中创建了该按钮, 并将它放置到 applet 上。

本例中重写了 action 方法。程序首先判断是否由 showFrame 按钮产生了该事件。如果是, 就利用下面的条件来判断框架对象 f 是否不为空:

```
f != null
```

如果该条件为 true, 那么就使用 hide 方法将框架 f 隐藏起来, 然后再使用 dispose 方法将其所占有的资源释放。

下面的语句将调用 DemoFrame 构造函数来创建对象 f:

```
f = new DemoFrame( s );
```

该框架通过 resize 方法重新设置大小。下面的语句是调用 Frame 的 setResizable 方法, 参数的值为 false:

```
f.setResizable( false );
```

setResizable 方法的功能是设置一个框架是否允许用户调整其大小。默认时框架的大小是可调整的, 除非使用 setResizable 方法将其设置为 false。show 方法用于显示框架。

DemoFrame 类是 Frame 类的扩展类。DemoFrame 类包含 4 个 Button 对象。它的构造函数带有一个字符串参数。下面的语句将调用超类 (即 Frame) 的构造函数, 并传入参数值 s:

```
super( s );
```

该构造函数声明了各个按钮对象, 并将它们添加到框架上。默认的布局管理器是 BorderLayout, 所以在添加按钮时要指定每个按钮的位置。

本例中重写了 handleEvent 方法。首先利用下面的条件, 判断是否为 WINDOW\_DESTROY 事件:

```
e.id == Event.WINDOW_DESTROY
```

该语句判断 e.id 的值是否为 Event 常量 WINDOW\_DESTROY。当使用鼠标点击窗口上的退出图标时,

就会产生 WINDOW\_DESTROY 事件。在 `handleEvent` 方法中，调用 `hide` 方法和 `dispose` 方法来隐藏和消除框架。如果由 `handleEvent` 接收的事件不是 WINDOW\_DESTROY 事件，那么就会调用超类的 `handleEvent` 方法，将该事件传递到适当的事件处理方法中。

在 `DemoFrame` 中重写了 `action` 方法。当使用鼠标点击框架上的一个按钮时，所产生的事件就会由 `DemoFrame` 的 `handleEvent` 方法传递到 `DemoFrame` 的 `action` 方法中。在 `action` 方法中，首先判断按下的是哪一个按钮，然后使用 `setBackground` 方法，将框架的背景转换成相应的颜色。最后调用 `repaint` 方法，利用转换后的颜色刷新框架的背景。

`Frame` 类还可用于创建基于框架的应用程序，它为应用程序 GUI 的创建提供了空间。图 11.14 的程序就是一个基于框架的应用程序，该程序等价于图 11.13 的 applet。

---

```
1    // Fig. 11.14: MyFrame3.java
2    // Creating a Frame-based application.
3    import java.awt.*;
4
5    public class MyFrame3 extends Frame {
6        private Button showFrame;
7        private DemoFrame f;
8        private int number;
9        private String frameTitle, title;
10
11        // default constructor
12        public MyFrame3()
13        {    this( "Frame" );    }
14
15        public MyFrame3( String t )
16        {
17            super( "Application" );
18            String s = "Click here to see the Frame";
19
20            number = 0;
21            title = t;
22            showFrame = new Button( s );
23            add( "South", showFrame );
24
25            resize( 300, 100 );
26            show();
27        }
28
29        public boolean handleEvent( Event e )
30        {
31            if ( e.id == Event.WINDOW_DESTROY ) {
32                removeFrame( this );
33                System.exit( 0 );    // terminate program
34                return true;
35            }
36
37            return super.handleEvent( e );
38        }
39
40        public boolean action( Event e, Object o )
41        {
```

---



```
42         if ( e.target == showFrame ) {
43
44             if ( f != null )
45                 removeFrame( f );
46
47             number++;
48             frameTitle = title + " " +
49                 String.valueOf( number );
50
51             f = new DemoFrame( frameTitle );
52         }
53
54         return true;
55     }
56
57     public void removeFrame( Frame w )
58     {
59         w.hide();        // hide frame
60         w.dispose();     // free resources
61     }
62
63     public static void main( String args[] )
64     {
65         MyFrame3 myself;
66
67         if ( args.length == 0 )
68             myself = new MyFrame3();
69         else
70             myself = new MyFrame3( args[0] );
71     }
72 }
73
74 class DemoFrame extends Frame {
75     private Button a, b, c, d;
76
77     public DemoFrame( String s )
78     {
79         super( s );
80         a = new Button( "yellow" );
81         b = new Button( "red" );
82         c = new Button( "blue" );
83         d = new Button( "green" );
84
85         // default layout is border
86         add( "North", a );
87         add( "East", b );
88         add( "South", c );
89         add( "West", d );
90
91         resize( 200, 200 );
92         show();    // display frame
93     }
94
95     public boolean handleEvent( Event e )
```

```

96      {
97          if ( e.id == Event.ACTION_EVENT ) {
98              action( e, e.arg );
99              return true;
100          }
101          else if ( e.id == Event.WINDOW_DESTROY ) {
102              hide();      // hide frame
103              dispose();  // free resources
104              return true;
105          }
106
107          return super.handleEvent( e );
108      }
109
110      public boolean action( Event e, Object o )
111      {
112          if ( e.target == c )
113              setBackground( Color.blue );
114          else if ( e.target == b )
115              setBackground( Color.red );
116          else if ( e.target == d )
117              setBackground( Color.green );
118          else // a button
119              setBackground( Color.yellow );
120
121          repaint();
122          return true;
123      }
124  }

```

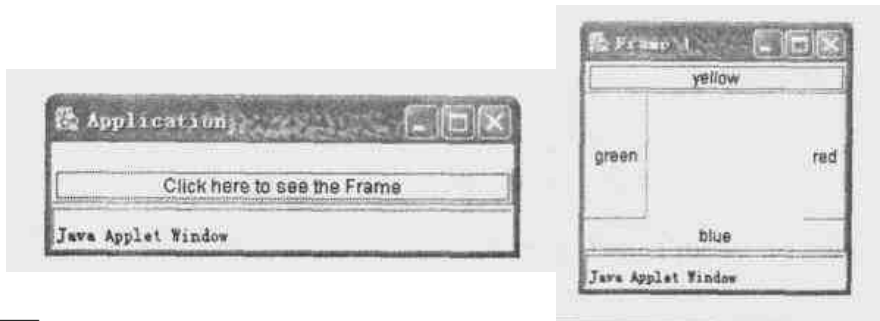


图 11.14 创建基于 Frame 的应用程序

图 11.14 中的 DemoFrame 类和图 11.13 中的 DemoFrame 类实质上是一样的。DemoFrame 类包含 4 个按钮对象，按照 BorderLayout 的方式放置。当使用鼠标点击某个按钮时，框架的背景就转换成相应的颜色。

MyFrame3 类含有两个构造函数。第一个构造函数是默认的，该构造函数通过下面的语句来调用其他的构造函数：

```
this ( " Frame " );
```

第二个构造函数带有一个字符串参数。字符串“Application”作为参数传入超类的构造函数，成为框架的标题栏文本。本例中创建了两个实例变量 number 和 title 以及一个 Button 对象，并将该按钮对象放到了框架的南区。然后重新设置框架的大小并将其显示出来。



**软件工程视点 11.3**

使用菜单可以减少用户所看到的组件的数目，进而简化 GUI 的结构。

图 11.16 的程序创建了一个简单的记事本应用程序，这个基于框架的应用程序由一个框架、一个文本区域和一个菜单栏组成。菜单栏中包含一个菜单，该菜单包含三个菜单项，其中每一个都表示一种不同的字体。当用户选择一个菜单项时，文本区域中的文本就转变成相应的字体。

```
1 // Fig. 11.16: ScratchPad.java
2 // Incorporating menus into a Frame-based application.
3 import java.awt.*;
4
5 public class ScratchPad extends Frame {
6     private TextArea t;
7     private Font f;
8
9     public ScratchPad()
10    {
11        super( "ScratchPad Application" );
12
13        t = new TextArea();
14        add( "Center", t );
15
16        f = new Font( "TimesRoman", Font.PLAIN, 12 );
17        setFont( f );
18
19        // create menubar
20        MenuBar bar = new MenuBar();
21
22        // create font menu
23        Menu fontMenu = new Menu( "Font" );
24
25        // add some items to the menu
26        fontMenu.add( "Times Roman" );
27        fontMenu.add( "Courier" );
28        fontMenu.add( "Helvetica" );
29
30        // add menu to menu bar
31        bar.add( fontMenu );
32
33        // set the menubar for the frame
34        setMenuBar( bar );
35
36        resize( 300, 200 );
37        show();
38    }
39
40    public boolean handleEvent( Event e )
41    {
42        if ( e.id == Event.WINDOW_DESTROY ) {
43            hide(); // hide frame
44            dispose(); // free resources
45            System.exit( 0 ); // terminate
46            return true;
47        }
48    }
```

```

49         return super.handleEvent( e );
50     }
51
52     public boolean action( Event e, Object o )
53     {
54         if ( e.target instanceof MenuItem ) {
55
56             if ( e.arg.equals( "Times Roman" ) )
57                 f = new Font( "TimesRoman", Font.PLAIN, 12 );
58             else if ( e.arg.equals( "Courier" ) )
59                 f = new Font( "Courier", Font.PLAIN, 12 );
60             else // Helvetica
61                 f = new Font( "Helvetica", Font.PLAIN, 12 );
62
63             t.setFont( f );
64         }
65
66         return true;
67     }
68
69     public static void main( String args[] )
70     {
71         ScratchPad e;
72
73         e = new ScratchPad();
74     }
75 }

```

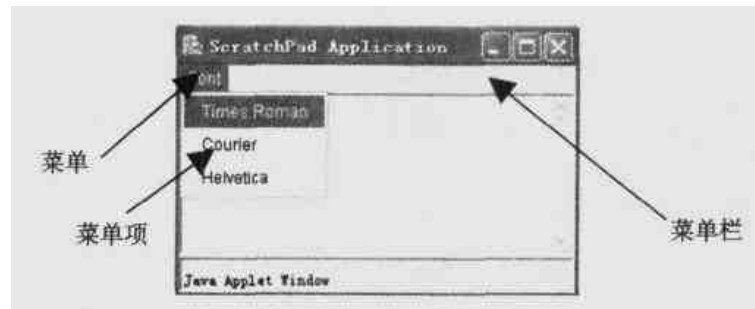


图 11.16 将菜单加入到基于框架的应用程序中

ScratchPad类创建了一个TextArea对象和一个Font对象。该类有一个构造函数，首先调用超类的构造函数来设置标签，然后创建TextArea对象，并将它添加到框架的中区。这里另外还创建了一个Font对象f。下面的语句使用了MenuBar构造函数来创建菜单栏对象：

```
MenuBar bar =newMenuBar ( );
```

Font菜单由下面的语句创建：

```
Menu fontMenu =new Menu ( " Font " ) ;
```

构造函数Menu的参数是一个字符串，表示菜单的名称。下面的语句创建了3个菜单项，并把它们添加到fontMenu中：

```
fontMenu.add ( new MenuItem ( " Times Roman " ) );
fontMenu.add ( new MenuItem ( " Courier " ) );
fontMenu.add ( new MenuItem ( " Helvetica " ) );

```

菜单项是利用 `MenuItem` 构造函数创建的。`MenuItem` 构造函数的参数是一个字符串, 即该菜单项的名称。`Menu` 类的 `add` 方法用于将一个菜单项添加到菜单中。下面的语句用于将 `fontMenu` 添加到菜单栏上:

```
bar.add( fontMenu );
```

通过 `MenuBar` 类的 `add` 方法可以将一个菜单添加到菜单栏上。下面的语句是使用 `Frame` 类的 `setMenuBar` 方法将菜单栏添加到框架上:

```
setMenuBar ( bar );
```

接着设置框架的大小, 并将其显示出来。

#### 常见编程错误 11.6

如果忘记使用 `Frame` 类的 `setMenuBar` 方法来设置菜单栏, 那么该菜单栏在框架上就显示不出来。

#### 常见的编程错误 11.7

如果试图在 applet 中使用菜单, 则会产生编译错误。

程序重写了 `handleEvent` 方法。首先判断产生的事件是否为 `WINDOW_DESTROY` 事件。如果是, 就将框架从屏幕上消除, 然后调用 `exit` 方法终止程序。如果是其他的事件, 就将该事件传递给超类的 `handleEvent` 方法。

同时, 程序还重写了 `action` 方法。首先利用下面的条件, 判断产生事件的是否为菜单项:

```
e.target instanceof MenuItem
```

如果是, 就使用 `if/else` 语句判断产生事件的是哪一个菜单项。`Event` 的实例变量 `arg` 中包含的是一个字符串, 即选中的菜单项的名称。根据选中的菜单项来创建一个 `Font` 对象, 然后利用 `setFont` 方法修改文本区域的字体。

用于判断每个菜单项的条件都是使用 `equals` 方法, 例如:

```
e.arg.equals( " Courier " )
```

如果每个菜单项的名称都是惟一的, 那么利用这种方法进行判断就是可行的。如果多个菜单项具有同样的名称 (可能是在不同的菜单中), 那么就应该使用下面的条件进行判断:

```
e.target == item2
```

其中 `item2` 是一个 `MenuItem` 对象。否则, 如果两个菜单项具有相同的名称, 那么它们将完成同样的动作。

菜单是我们所介绍的可能产生事件的最后一种组件, 图 11.17 中列出了 GUI 组件的所有事件。

#### 常见编程错误 11.8

将 `MenuItem` 或 `CheckboxMenuItem` 中的 “I” 写成小写的 “i” 会产生语法错误。

#### 常见编程错误 11.9

将类名称 `MenuBar` 中的 “B” 写成小写的 “b” 会产生语法错误。

#### 常见编程错误 11.10

将类名称 `CheckboxMenuItem` 中的 “b” 写成大写的 “B” 会产生语法错误。

**常见编程错误 11.11**

忘记向菜单中添加菜单项会产生运行时的逻辑错误, 这种情况下菜单项将不会出现在菜单中。

**常见编程错误 11.12**

在菜单中使用 Checkbox 组件而不是使用 CheckboxMenuItem 组件将会产生语法错误。

| 组件       | 产生事件的动作 | arg 实例变量值 | arg 数据类型 | 事件处理方法      |
|----------|---------|-----------|----------|-------------|
| 按钮       | 按下按钮    | 按钮标签      | String   | action      |
| 画板       | 无       | 无         | 无        | 无           |
| 复选框或单选按钮 | 点击复选框   | 复选框的状态    | Boolean  | action      |
| 选择按钮     | 选择选项    | 选中的选项     | String   | action      |
| 标签       | 无       | 无         | 无        | 无           |
| 列表       | 双击选项    | 选中的选项     | String   | action      |
| 菜单项      | 选择选项    | 选中的选项     | String   | action      |
| 面板       | 无       | 无         | 无        | 无           |
| 滚动条      | 点击滚动条   | 滚动条的值     | Integer  | handleEvent |
| 文本区域     | 无       | 无         | 无        | 无           |
| 文本字段     | 按下回车键   | 文本字段中的文本  | String   | action      |

图 11.17 GUI 组件的各个事件

图 11.18 的程序是基于框架的应用程序, 它要比图 11.16 的程序功能有所增强, 其中增加了颜色选项和使文本区域只读的选项。该程序说明了如何使用子菜单。

在这个例子中, 声明了一个 TextArea 对象、一个 Font 对象、一个 Color 对象和一个 CheckboxMenuItem 对象。在该类的构造函数中, 首先调用超类的构造函数来设置框架的标题; 接下来创建 TextArea 对象, 将它添加到框架中; 然后创建 Font 对象, 并将 TextArea 对象设置成该字体; 接着建立菜单, 设置菜单栏以及框架的大小; 最后显示框架。

```

1    // Fig. 11.18: Scratch2.java
2    // Using submenus in a menu.
3    import java.awt.*;
4
5    public class Scratch2 extends Frame {
6        private TextArea t;
7        private Font f;
8        private Color c;
9        private CheckboxMenuItem x;
10
11    public Scratch2()
12    {
13        super( "ScratchPad Application 2" );
14
15        t = new TextArea();
16        add( "Center", t );
17
18        f = new Font( "TimesRoman", Font.PLAIN, 12 );
19        t.setFont( f );
20
21        t.setForeground( Color.black );
22
23        // create menubar
24        MenuBar bar = new MenuBar();

```

```
25
26     // create menus
27     Menu viewMenu = new Menu( "View" );
28     Menu fontMenu = new Menu( "Font" );
29     Menu colorMenu = new Menu( "Color" );
30
31     // build color menu
32     colorMenu.add( new MenuItem( "Black" ) );
33     colorMenu.add( new MenuItem( "Blue" ) );
34
35     // build font menu
36     fontMenu.add( new MenuItem( "Times Roman" ) );
37     fontMenu.add( new MenuItem( "Courier" ) );
38
39     // build view menu
40     viewMenu.add( fontMenu );
41     viewMenu.add( new MenuItem( "-" ) );
42     viewMenu.add( colorMenu );
43     viewMenu.add( new MenuItem( "-" ) );
44
45     x = new CheckboxMenuItem( "Read-only" );
46     viewMenu.add( x );
47     x.setState( false );
48
49     // add menu to menu bar
50     bar.add( viewMenu );
51
52     // set the menubar for the frame
53     setMenuBar( bar );
54
55     resize( 300, 200 );
56     show();
57 }
58
59 public boolean handleEvent( Event e )
60 {
61     if ( e.id == Event.WINDOW_DESTROY ) {
62         hide();          // hide frame
63         dispose();       // free resources
64         System.exit( 0 ); // terminate
65         return true;
66     }
67
68     return super.handleEvent( e );
69 }
70
71 public boolean action( Event e, Object o )
72 {
73     if ( e.target instanceof MenuItem ) {
74
75         if ( e.arg.equals( "Times Roman" ) )
76             f = new Font( "TimesRoman", Font.PLAIN, 12 );
77         else if ( e.arg.equals( "Courier" ) )
78             f = new Font( "Courier", Font.PLAIN, 12 );
79         else if ( e.arg.equals( "Black" ) )
80             c = Color.black;
```



```

81         else if ( e.arg.equals( "Blue" ) )
82             c = Color.blue;
83         else if ( e.arg.equals( x.getLabel() ) )
84             t.setEditable( !x.getState() );
85
86         t.setForeground( c );
87         t.setFont( f );
88     }
89
90     return true;
91 }
92
93 public static void main( String args[] )
94 {
95     Scratch2 e;
96
97     e = new Scratch2();
98 }
99 }

```

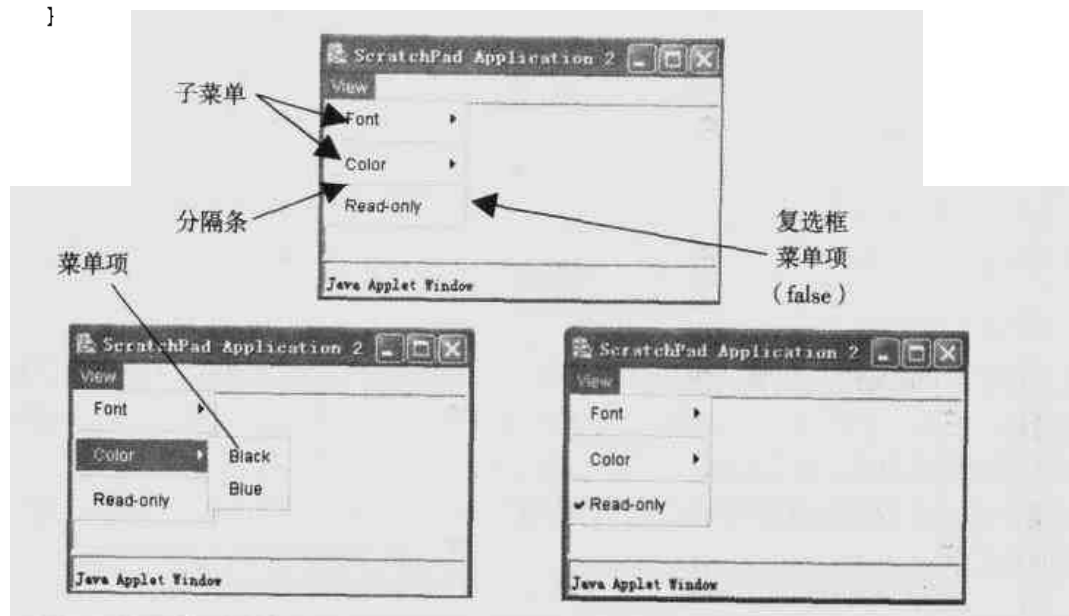


图 11.18 在菜单中使用子菜单

菜单是通过调用 `MenuBar` 构造函数而创建的, 可以分别使用下面的语句创建 3 个菜单:

```

viewMenu = new Menu ( " View " );
fontMenu = new Menu ( " Font " );
colorMenu = new Menu ( " Color " );

```

`colorMenu` 对象和 `fontMenu` 对象中分别包含两个菜单项。利用下面的语句将 `fontMenu` 对象添加到 `viewMenu` 中:

```
viewMenu.add ( fontMenu );
```

当把 `fontMenu` 添加到 `viewMenu` 中时, 就建立了一个子菜单。选择该子菜单, 就可以显示子菜单中的各个菜单项。注意, 子菜单上有一个向右的箭头, 表示此处有子菜单项。

下面的语句是利用特殊形式的 `MenuItem` 构造函数创建一个分隔条——可以将菜单项显式分开的一条线:

```
viewMenu . add ( new MenuItem ( "-" ) );
```

将 MenuItem 构造函数的参数设置为连字符“-”，就可以创建一个分隔条。使用鼠标点击分隔条是会产生任何事件的。然后将 colorMenu 和另一个分隔条也添加到 viewMenu 中。

下面的语句是利用 CheckboxMenuItem 构造函数创建一个复选框菜单项：

```
x = new CheckboxMenuItem ( " Read-only " );
```

可以通过下面的语句将该复选框菜单项添加到 viewMenu 中：

```
viewMenu.add ( x ) ;
```

下面的语句用于将复选框菜单项的状态设置成 false：

```
x.setState ( false );
```

可以利用 CheckboxMenuItem 的 setState 方法设置复选框菜单项的布尔值，当将复选框菜单项设置成 false 时，该菜单项左边的选中标记就会消失。

ViewMenu 将由 add 方法添加到菜单栏上。然后在框架上设置菜单栏，最后设置框架的大小，并将其显示出来。

本例中重写了 handleEvent 方法。首先判断产生的事件是否为 WINDOW\_DESTROY 事件。如果是，就将该框架从屏幕上隐藏并消除，然后调用 exit 方法终止程序。如果是其他事件，就将其传送到超类的 handleEvent 方法中。

本例中重写了 action 方法。首先利用下面的条件判断产生事件的是否为菜单项：

```
e.target instanceof MenuItem
```

如果是，就使用 if/else 语句判断产生事件的是哪一个菜单项。Event 实例变量 arg 中包含的是选中菜单项的名称。根据选中的菜单项，创建一个 Font 对象和 Color 对象。然后使用 setForeground 方法修改文本区域的背景颜色，并使用 setFont 方法将文本区域的字体修改为 f。

如果产生事件的是复选框菜单项，那么就用 CheckboxMenuItem 的 getState 方法获取复选框菜单项的当前状态：

```
t.setEditable( ! x.getState ( ) );
```

如果 getState 的返回值为 true，那么该文本区域的 setEditable 方法的输入值就为 false——反之亦然。

## 11.8 对话框

对话框是一个带有标题栏的无边界窗口，通常用于接收用户的信息或向用户显示信息。对话框用 Dialog 类创建，而 Dialog 类是从 Window 类继承的。

对话框分为模态的或非模态的。当一个模态的对话框打开时，不允许访问应用程序中的其他窗口，直到该对话框关闭。而当一个非模态的对话框打开时，用户仍然可以访问其他窗口。对话框的默认布局管理器是 BorderLayout。

图 11.19 中程序的功能是，当用户选择 Help 菜单的 About... 菜单项时显示出一个对话框。该对话框是非模态的，这样，当显示对话框时，用户也能访问框架及它的菜单。该对话框将显示文本“Java How To Program”及一个 OK 按钮。当对话框出现时，About... 菜单就变成灰色——即用鼠标点击它时不会产生事件。

在 DialogBoxes 类中, 声明了一个 About 对象和两个 MenuItem 对象。这个类的构造函数首先调用超类的构造函数, 然后创建菜单栏, 接着创建 File 和 Help 菜单, 将菜单添加到菜单栏上, 再设置菜单栏。最后设置框架的大小, 并显示框架。

下面的语句是通过 MenuBar 的 setHelpMenu 方法将 help Menu 添加到菜单栏上:

```
bar.setHelpMenu ( helpMenu );
```

当设置 help Menu 时, 它将自动出现在菜单栏的最右边。注意, File 菜单实际是在 Help 菜单之后添加的, 但 Help 菜单仍然位于最右边。

```

1      // Fig. 11.19: DialogBoxes.java
2      // Demonstrating modeless Dialog boxes.
3      import java.awt.*;
4
5      public class DialogBoxes extends Frame {
6          private About a;
7          private MenuItem item, item2;
8
9          public DialogBoxes()
10         {
11             super( "Dialog boxes" );
12
13             MenuBar bar = new MenuBar();
14             Menu helpMenu = new Menu( "Help" );
15             Menu fileMenu = new Menu( "File" );
16
17             item = new MenuItem( "About..." );
18             item2 = new MenuItem( "Exit" );
19
20             helpMenu.add( item );
21             fileMenu.add( item2 );
22
23             bar.setHelpMenu( helpMenu );
24             bar.add( fileMenu );
25             bar.add( helpMenu );
26
27             setMenuBar( bar );
28             resize( 200, 100 );
29             show();
30         }
31
32         public static void main( String args[ ] )
33         {
34             DialogBoxes d;
35
36             d = new DialogBoxes();
37         }
38
39         public boolean action( Event e, Object o )
40         {
41             if ( e.target instanceof MenuItem )
42
43                 if ( e.arg.equals( item.getLabel() ) ) {
44                     a = new About( this );

```

```
45         setItemState( false );
46     }
47     else        // Exit menu item
48         removeFrame();
49
50     return true;
51 }
52
53 public boolean handleEvent( Event e )
54 {
55     if ( e.id == Event.WINDOW_DESTROY ) {
56         removeFrame();
57         return true;
58     }
59
60     return super.handleEvent( e );
61 }
62
63 // setItemState is programmer-defined
64 public void setItemState( boolean state )
65 {
66     if ( state == true )
67         item.enable();
68     else
69         item.disable();
70 }
71
72 // removeFrame is programmer-defined
73 public void removeFrame()
74 {
75     hide();
76     dispose();
77     System.exit( 0 );
78 }
79 }
80
81 class About extends Dialog {
82     private Button b;
83     private Label l;
84     private Panel p, p2;
85     private DialogBoxes parent;
86
87     public About( Frame f )
88     {
89         super( f, "About", false );
90         parent = ( DialogBoxes ) f;
91
92         b = new Button( "Ok" );
93         p = new Panel();
94         p2 = new Panel();
95         l = new Label( "Java How To Program" );
96
97         p.add( l );
98         p2.add( b );
99
100        add( "Center", p );
```

```

101         add( "South", p2 );
102         resize( 200, 100 );
103         show();
104     }
105
106     public boolean handleEvent( Event e )
107     {
108         if ( e.id == Event.WINDOW_DESTROY ) {
109             removeDialog();
110             return true;
111         }
112
113         return super.handleEvent( e );
114     }
115
116     public boolean action( Event e, Object o )
117     {
118         if ( e.target == b )
119             removeDialog();
120
121         return true;
122     }
123
124     // removeDialog is user-defined
125     public void removeDialog()
126     {
127         hide();
128         dispose();
129         parent.setItemState( true );
130     }
131 }

```



图 11.19 创建一个非模态的对话框

程序中重写了 action 方法。如果产生事件的是 About... 菜单项, 那么就执行下面的语句:

```

a = new About( this );
setItemState( false );

```

首先调用 About 构造函数来创建 About 对话框对象 a，其输入参数是当前的 Frame 对象 this。通过向对话框构造函数输入一个框架对象（比如 this）参数，可以将对话框连接到一个框架上。setItemState 方法使用 MenuItem 的 enable 方法和 disable 方法，可以将 About... 菜单项设置为激活的或未激活的。如果一个菜单项是激活的，那么选择该菜单项会产生一个 action 事件。如果菜单项是未激活的，那么该菜单项就会显示为灰色——选中显示为灰色的菜单是会产生任何事件的。

程序中重写了 handleEvent 方法。首先判断是否是 WINDOW\_DESTROY 事件，如果是，就调用 removeFrame 方法。removeFrame 方法的功能是先隐藏框架，然后调用 dispose 方法消除框架，最后调用 exit 方法来终止程序。如果是其他的事件，则将其发送到超类的 handleEvent 方法中。

如果要建立一个对话框，应该扩展 Dialog 类。与 Frame 类一样，Dialog 类本身的功能并不够用。About 类就是 Dialog 类的扩展。

在 About 类中声明了五个对象——一个 Button 对象、一个 Label 对象、两个 Panel 对象和一个 DialogBoxes 对象（即 DialogBoxes 应用程序类的对象）。DialogBoxes 的构造函数带有一个 Frame 参数，即该对话框的父容器。对话框不能处理的事件将发送到该对话框的父框架中。

下面的语句是调用超类的构造函数：

```
super ( f, " About " , false ) ;
```

该超类构造函数带有 3 个参数——父框架名、标题名及 1 个布尔值（标识该对话框是否为模态的）。如果第三个参数的值为 false，则表示对话框是非模态的。

About 对象的父框架由下面的语句创建：

```
parent = ( DialogBoxes ) f;
```

About 类可以通过 parent 与父框架的 public 方法进行交互操作。接着创建 Button 对象、Panel 对象和 Label 对象。Label 对象将添加到一个面板上，而 Button 对象则添加到另一个面板上。这两个面板都将添加到对话框上——一个位于中区，另一个位于南区。然后设置对话框的大小，并显示对话框。与框架一样，对话框没有默认的初始大小值，而且是不可见的。必须利用 resize 方法来设置对话框的大小，并使用 show 方法显示对话框。

#### 常见编程错误 11.13

在使用对话框时忘记调用 resize 方法会产生运行时的逻辑错误。因为对话框没有默认的初始大小值，所以它通常是不可见的。

#### 常见编程错误 11.14

忘记调用 show 方法会产生运行时的逻辑错误。因为对话框默认是隐藏的，所以当前未显示。

程序中重写了 action 方法，当用户按下 Ok 按钮时，就调用 removeDialog 方法。removeDialog 方法的功能是调用 hide 方法和 dispose 方法，将该对话框从屏幕上清除，然后再调用 parent 的 setItemState 方法。setItemState 的输入值是 true，因此将 About... 菜单项设置为激活的。

程序中也重写了 handleEvent 方法。首先判断产生的事件是否为 WINDOW\_DESTROY 事件。如果是，就调用 removeDialog 方法。如果是其他事件，就由超类的 handleEvent 方法调用 action 方法。

java.awt 中包含一个特殊的对话框类 FileDialog，用于文件处理。FileDialog 类是从 Dialog 类继承的。文件对话框的“外观”是由窗口系统预定义的。文件对话框默认为模态的，其中包括两种类型——Open 文件对话框和 Save 文件对话框。如果想打开一个文件或保存一个文件，不必编写任何代码，我们将在第 15 章讨论有关文件处理的详细内容。

图 11.20 的程序显示了一个文件对话框，当选择 Open 菜单项或 Save 菜单项时，该文件对话框

就会出现。文件对话框带有一些预定义的功能,比如搜索目录树。

在DialogBoxes2类的构造函数中,首先调用超类的构造函数,然后创建菜单、设置菜单栏、设置框架的大小,最后显示框架。

程序中重写了handleEvent方法,如果产生的是WINDOW\_DESTROY事件,则隐藏和清除框架,并调用exit方法终止程序;如果是其他事件,则调用action方法。

这里重写了action方法。如果产生事件的是Open菜单项,那么就执行下面的语句:

```
m = new MyFiles ( this, FileDialog.LOAD );
```

构造函数MyFiles的输入参数为this和FileDialog.LOAD,静态常量FileDialog.SAVE和FileDialog.LOAD用于指定要创建的文件对话框的类型。

MyFiles类是FileDialog类的扩展类。MyFiles类包括一个构造函数,该函数带有一个Frame参数和一个int参数。使用下面这条语句来调用超类的构造函数:

```
super ( f, ( type == FileDialog.LOAD ? " Open ":"Save") + "Dialog", type );
```

第一个参数是该文件对话框的父框架,第二个参数是文件对话框的标题,第三个参数的值为FileDialog.SAVE或FileDialog.LOAD。这里使用了resize方法,不过resize方法的输入值对文件对话框的大小并没有影响。为了使文件对话框可见,必须调用show方法。

```

1 // Fig. 11.20: DialogBoxes2.java
2 // Demonstrating the "Open" and "Save" dialog boxes.
3 import java.awt.*;
4
5 public class DialogBoxes2 extends Frame
6 {
7     private MyFiles m;
8     private MenuItem item, item2;
9
10    public DialogBoxes2()
11    {
12        super( "Dialog boxes 2" );
13
14        MenuBar bar = new MenuBar();
15        Menu fileMenu = new Menu( "File" );
16
17        item = new MenuItem( "Open..." );
18        item2 = new MenuItem( "Save As ..." );
19
20        fileMenu.add( item );
21        fileMenu.add( item2 );
22
23        bar.add( fileMenu );
24        setMenuBar( bar );
25        resize( 200, 100);
26        show();
27    }
28
29    public static void main( String args[] )
30    {
31        DialogBoxes2 d = new DialogBoxes2();
32    }
33
34    public boolean action( Event e, Object o )

```

```

35     {
36         if ( e.target instanceof MenuItem )
37         {
38             if ( e.arg.equals( item.getLabel() ) )
39                 m = new MyFiles( this, FileDialog.LOAD );
40             else if ( e.arg.equals( item2.getLabel() ) )
41                 m = new MyFiles( this, FileDialog.SAVE );
42         }
43         return true;
44     }
45
46     public boolean handleEvent( Event e )
47     {
48         if ( e.id == Event.WINDOW_DESTROY ) {
49             hide();
50             dispose();
51             System.exit( 0 );
52             return true;
53         }
54
55         return super.handleEvent( e );
56     }
57 }
58
59 class MyFiles extends FileDialog
60 {
61     public MyFiles( Frame f, int type )
62     {
63         super( f, ( type == FileDialog.LOAD ? "Open "
64             : "Save " ) + "Dialog", type );
65
66         resize( 400, 250 );
67         show();
68     }
69 }

```

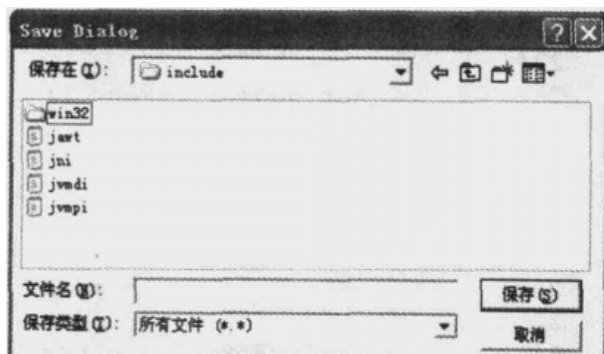
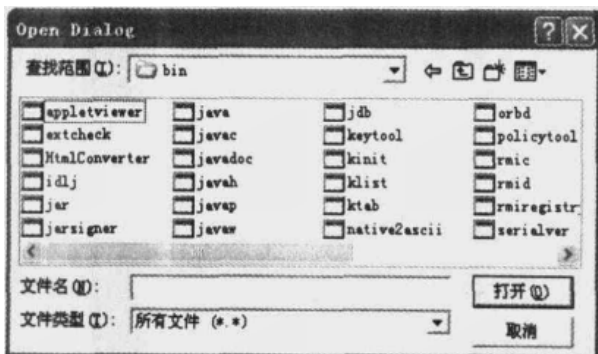
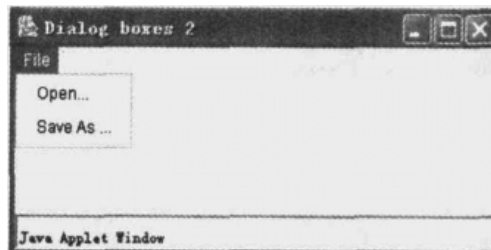


图 11.20 显示一个文件对话框



## 常见编程错误 11.15

使用 `FileDialog` 对象时忘记调用 `show` 方法会产生运行时的逻辑错误，文件对话框将无法显示出来。

## 11.9 高级布局管理器

在上一章中，我们介绍了 3 种布局管理器：`FlowLayout`、`BorderLayout` 和 `GridLayout`。在以下几节中，我们将讨论另外两种布局管理器（如图 11.21 所示）。

| 布局管理器                      | 描述                                                                                          |
|----------------------------|---------------------------------------------------------------------------------------------|
| <code>CardLayout</code>    | 该布局管理器是将各组件排放到一个卡片盒中，每个卡片上可以使用任何布局管理器，只有最上面的卡片是可见的                                          |
| <code>GridBagLayout</code> | 该布局管理器类似于 <code>GridLayout</code> ；但与 <code>GridLayout</code> 不同的是，每个组件的大小可以不一样，而且可以按任何顺序添加 |
| <code>None</code>          | 不使用布局管理器，每个组件都是手工排放                                                                         |
| 用户定义                       | 由程序员创建的定制布局管理器                                                                              |

图 11.21 高级布局管理器

### 11.10 CardLayout 布局管理器

`CardLayout` 布局管理器是将各个组件作为卡片而排放到一个“卡片盒”中，只有最上面的卡片可见。可以通过函数调用将这盒卡片中的任何一张移到最上面来，每张卡片通常都是一个容器（比如面板），在卡片上可以使用任何布局管理器。`CardLayout` 类是从 `Object` 类继承的，并提供了 `LayoutManager` 接口，我们将在第 13 章进一步讨论这些接口。

图 11.22 的程序创建了五个面板。其中一个面板使用了 `CardLayout` 布局管理器，它可以控制哪一张卡片位于最上面；另外三个面板就作为卡片。该程序仅仅显示了一个 GUI，没有执行任何操作。

```

1      // Fig. 11.22: CardDeck.java
2      // Demonstrating CardLayout.
3      import java.applet.Applet;
4      import java.awt.*;
5
6      public class CardDeck extends Applet {
7          private Canvas c;
8          private CardLayout cardManager;
9          private Panel deck, p1, p2, p3, p4;
10         private Button b1, b2, b3,
11                 prevButton, nextButton,
12                 lastButton, firstButton;
13
14         public void init()
15         {
16             setLayout( new BorderLayout() );    // applet
17
18             b1 = new Button( "card one  " );
19             b2 = new Button( "card two  " );
20             b3 = new Button( "card three" );
21
22             // create and customize a canvas

```

```
23      c = new Canvas();
24      c.setBackground( Color.green );
25      c.resize( 80, 80 );
26
27      deck = new Panel();
28      cardManager = new CardLayout(); // instantiate object
29      deck.setLayout( cardManager ); // set cardLayout
30      add( "East", deck );
31
32      // use the default layout for p1
33      p1 = new Panel();
34      p1.add( b1 ); // add a button
35      deck.add( b1.getLabel(), p1 ); // add card to deck
36
37      // set up second card
38      p2 = new Panel();
39      p2.add( b2 ); // add a button
40      p2.add( c ); // add a canvas
41      deck.add( b2.getLabel(), p2 ); // add card to deck
42
43      // set up last card
44      p3 = new Panel();
45      p3.setLayout( new BorderLayout() ); // set layout
46      p3.add( "North", new Button( "North Pole" ) );
47      p3.add( "West", new Button( "The West" ) );
48      p3.add( "East", new Button( "Far East" ) );
49      p3.add( "South", new Button( "South Pole" ) );
50      p3.add( "Center", b3 );
51      deck.add( b3.getLabel(), p3 ); // add card to deck
52
53      // create and layout panel that will control deck
54      p4 = new Panel();
55      p4.setLayout( new GridLayout( 2, 2 ) );
56      p4.add( firstButton = new Button( "First" ) );
57      p4.add( nextButton = new Button( "Next" ) );
58      p4.add( prevButton = new Button( "Previous" ) );
59      p4.add( lastButton = new Button( "Last" ) );
60      add( "West", p4 );
61  }
62
63  public boolean action( Event e, Object o )
64  {
65      if ( e.target == firstButton )
66          cardManager.first( deck ); // show first card
67      else if ( e.target == prevButton )
68          cardManager.previous( deck ); // show previous card
69      else if ( e.target == nextButton )
70          cardManager.next( deck ); // show next card
71      else if ( e.target == lastButton )
72          cardManager.last( deck ); // show last card
73
74      return true;
75  }
76 }
```



图 11.22 CardLayout 布局管理器

在 CardDeck 类中, 声明了 Panel 对象、Button 对象和 Canvas 对象。下面的语句用于声明一个 CardLayout 对象:

```
private CardLayout cardManager;
```

有了 CardLayout 对象, 就可以使用 CardLayout 类的方法遍历所有的“卡片”了。CardDeck 类包含两个方法——init 和 action。

在 init 方法中创建了 GUI。将 applet 的布局管理器设置成 BorderLayout, 接着创建三个 Button 对象和一个 Canvas 对象。将该画板的背景设置成绿色, 并设置了大小。下面的语句分别创建了 Panel 类的 deck 对象和 CardLayout 类的 cardManager 对象, 并将 deck 的布局管理器设置成 CardLayout:

```
deck = new Panel ( );
cardManager = new CardLayout ( );
deck.setLayout ( cardManager );
```

在创建每张卡片时, 都将其添加到 deck 中, 然后将 deck 面板添加到 applet 的东区。

下面的语句分别创建了面板 p1, 并将按钮 b1 添加到面板 p1 上, 然后再将面板 p1 添加到 deck 上:

```
p1 = new Panel ( );
p1.add ( b1 );
deck.add ( b1.getLabel ( ), p1 );
```

第一个参数用于标识已添加的组件, 第二个参数是要添加的卡片。使用类似的方法创建面板 p2 和 p3, 并添加到 deck 中。注意, p3 的布局管理器为 BorderLayout。

面板 p4 位于 applet 的西区, 其中包含了一些按钮, 用户可以利用这些按钮遍历所有的卡片。每个按钮的名称表示应显示哪一张卡片。当该程序运行时, 首先显示第一张添加的卡片 p1。

程序中重写了 action 方法, 首先判断点击的是 p4 的哪一个按钮。CardLayout 的 first、previous、next 和 last 方法用于显示卡片。first 方法用于显示添加到 deck 中的第一张卡片, previous 方法用于显示 deck 中的前一张卡片, next 方法用于显示 deck 中的下一张卡片, last 方法用于显示 deck 中的最后一张卡片。注意, 每个方法的输入参数都是 deck。

11.11 GridBagLayout 布局管理器

GridBagLayout是最复杂的预定义布局管理器。GridBagLayout布局管理器和GridLayout布局管理器类似，因为GridBagLayout也是将组件放置在网格中。不过GridBagLayout更灵活一些，各组件的大小可以不一样，并可以按任何顺序添加。

使用GridBagLayout的第一步是确定GUI的外观。这一步不需要任何编程，只需要显示GUI的一张画纸。接着在GUI上绘制网格，将各组件划分成行和列。初始行和列的坐标都从0开始，以便GridBagLayout能够恰当地安排各组件。GridBagLayout管理器使用行和列的坐标来准确地放置组件。图11.23说明了在GUI中如何划分行和列。

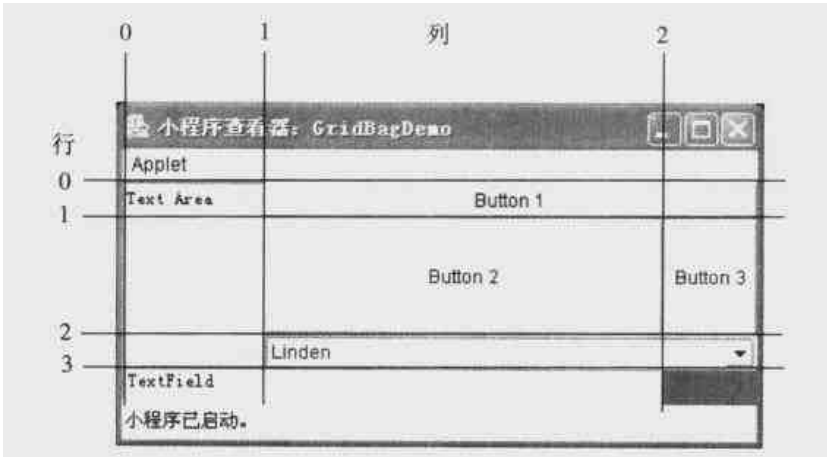


图 11.23 利用 GridBagLayout 设计 GUI

如果想使用GridBaglayout布局管理器,必须创建一个GridBagConstraints对象。GridBagConstraints对象用于确定如何使用GridBagLayout来对组件进行布局。图11.24中给出了几个重要的GridBagConstraints实例变量。

| GridBagConstraints 实例值 | 说明                                  |
|------------------------|-------------------------------------|
| gridx                  | 组件放置的列坐标                            |
| gridy                  | 组件放置的行坐标                            |
| gridwidth              | 组件所占的列数                             |
| gridheight             | 组件所占的行数                             |
| weightx                | 为构件分配的垂直方向的额外空间。若允许额外空间,则构件可以变得“更高” |
| weighty                | 为构件分配的水平方向的额外空间。若允许额外空间,则构件可以变得“更宽” |

图 11.24 GridBagConstraints 实例变量

gridx 和 gridy 变量用于确定组件的左上角位于什么位置 (坐标值), gridx 变量对应于列坐标, gridy 变量对应于行坐标。在图 11.23 中, 选择按钮 “Linden” 的 gridx 值为 1、gridy 值为 2。

gridwidth 变量用于确定组件所占的列数。在图 11.23 中, 选择按钮 “Linden” 占 2 列。gridheight 变量用于确定组件所占的行数。在图 11.23 中, 文本区域 TextArea 占 3 行。

当容器大小发生变化时, 可以使用 weightx 变量确定如何为组件分配垂直方向的额外空间; 如果该值为 0, 则表示组件不会在垂直方向上增长。weighty 变量用于确定如何为组件分配水平方向的额外空间; 如果该值为 0, 则表示组件不会在水平方向上增长。

## 常见的编程错误 11.16

如果 `weightx` 或 `weighty` 的值为负数, 则会产生逻辑错误。

在图 11.23 中, 看不出 `weightx` 和 `weighty` 的效果。但当容器的大小发生变化时, 这两个值就会起到作用。`weight` 值大的组件将比 `weight` 值小的组件占用更多空间。在后面的练习中, 我们将看到不同的 `weightx` 和 `weighty` 值会产生不同的效果。

组件的 `weight` 值应该不为 0, 否则各组件将在容器的中间“挤作一团”。图 11.25 显示了图 11.23 的 GUI, 其中所有的 `weight` 值都设为 0。

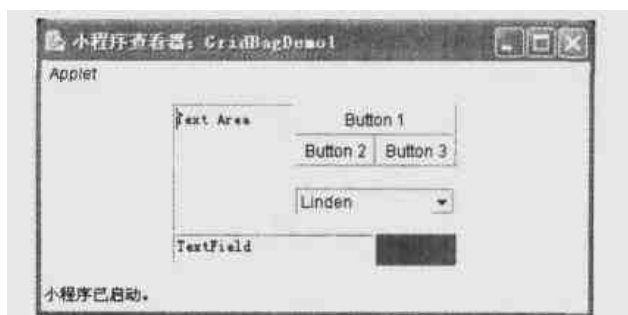


图 11.25 `weight` 值为 0 的 `GridBagLayout`

`GridBagConstraints` 实例变量 `fill` 用于确定占用多少组件区域。变量 `fill` 的值为下面的 `GridBagConstraints` 静态常量之一: `NONE`、`VERTICAL`、`HORIZONTAL` 或 `BOTH`, 默认值为 `GridBagConstraints.NONE`。

当组件没有占满整个区域时, 可以使用 `GridBagConstraints` 的实例变量 `anchor` 来确定将组件放置到该区域的什么位置。变量 `anchor` 的值是下列 `GridBagConstraints` 静态常量之一: `NORTH`、`NORTHEAST`、`EAST`、`SOUTHEAST`、`SOUTH`、`SOUTHWEST`、`WEST`、`NORTHWEST` 或 `CENTER`。默认值为 `GridBagConstraints.CENTER`。

图 11.26 的程序的功能, 是使用 `GridBagLayout` 管理器安排图 11.23 的 GUI 中的各组件。

```

1 // Fig. 11.26: GridBagDemo.java
2 // Demonstrating GridBagLayout.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class GridBagDemo extends Applet {
7     private Canvas c;
8     private Choice cb;
9     private TextArea ta;
10    private TextField tf;
11    private Button b1, b2, b3;
12    private GridBagLayout gbLayout;
13    private GridBagConstraints gbConstraints;
14
15    public void init()
16    {
17        gbLayout = new GridBagLayout();
18        setLayout( gbLayout ); // applet
19
20        // instantiate gridbag constraints
21        gbConstraints = new GridBagConstraints();

```

```

22
23     ta = new TextArea( "Text Area", 5, 10 );
24     cb = new Choice( );
25     cb.addItem( "Linden" );
26     cb.addItem( "Birch" );
27     cb.addItem( "Ceder" );
28     tf = new TextField( "TextField" );
29     b1 = new Button( "Button 1" );
30     b2 = new Button( "Button 2" );
31     b3 = new Button( "Button 3" );
32     c = new Canvas();
33     c.setBackground( Color.blue );
34     c.resize( 10, 5 );
35
36     // text area
37     // weightx and weighty are both 0: the default
38     // anchor for all components is CENTER: the default
39     gbConstraints.fill = GridBagConstraints.BOTH;
40     addComponent( ta, gbLayout, gbConstraints, 0, 0, 1, 3 );
41
42     // button b1
43     // weightx and weighty are both 0: the default
44     gbConstraints.fill = GridBagConstraints.HORIZONTAL;
45     addComponent( b1, gbLayout, gbConstraints, 0, 1, 2, 1 );
46
47     // choice button
48     // weightx and weighty are both 0: the default
49     // fill is HORIZONTAL
50     addComponent( cb, gbLayout, gbConstraints, 2, 1, 2, 1 );
51
52     // button b2
53     gbConstraints.weightx = 1000; // can grow taller
54     gbConstraints.weighty = 1;    // can grow wider
55     gbConstraints.fill = GridBagConstraints.BOTH;
56     addComponent( b2, gbLayout, gbConstraints, 1, 1, 1, 1 );
57
58     // button b3
59     // fill is BOTH
60     gbConstraints.weightx = 0;
61     gbConstraints.weighty = 0;
62     addComponent( b3, gbLayout, gbConstraints, 1, 2, 1, 1 );
63
64     // textfield
65     // weightx and weighty are both 0: fill is BOTH
66     addComponent( tf, gbLayout, gbConstraints, 3, 0, 2, 1 );
67
68     // canvas
69     // weightx and weighty are both 0: fill is BOTH
70     addComponent( c, gbLayout, gbConstraints, 3, 2, 1, 1 );
71 }
72
73 // addComponent is programmer defined
74 private void addComponent( Component c, GridBagLayout g,
75                           GridBagConstraints gc, int row,
76                           int column, int width, int height )
77 {

```

```

78         // set gridx and gridy
79         gc.gridx = column;
80         gc.gridy = row;
81
82         // set gridwidth and gridheight
83         gc.gridwidth = width;
84         gc.gridheight = height;
85
86         g.setConstraints( c, gc ); // set constraints
87         add( c ); // add component to applet
88     }
89 }

```

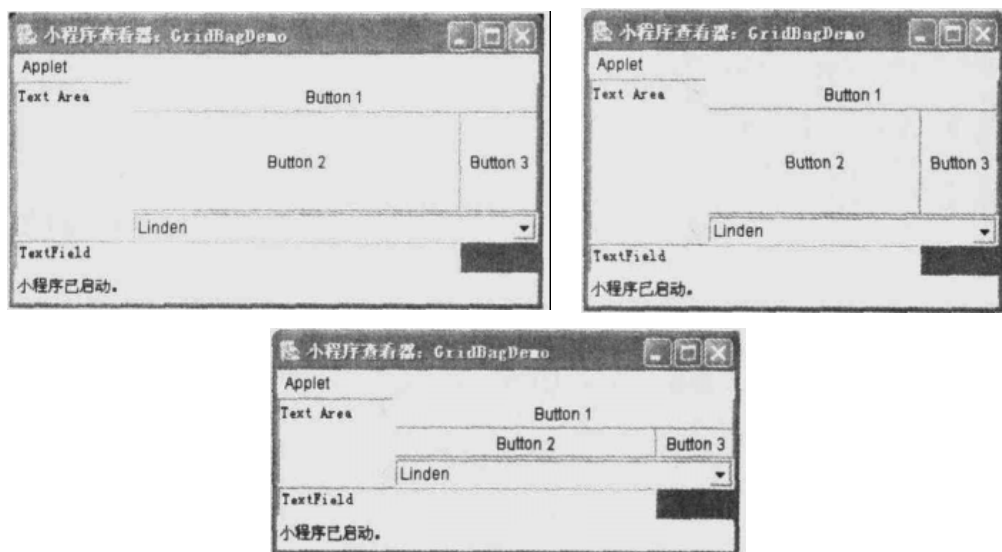


图 11.26 GridBagLayout 布局管理器

该GUI中包括三个按钮、一个文本区域、一个选择按钮、一个画板及一个文本字段，它所用的布局管理器是 GridBagLayout。下面的语句将调用 GridBagLayout 构造函数来创建 gblayout 对象，然后将 applet 的布局管理器设置成 gblayout：

```

gblayout = new GridBagLayout ( );
setLayout ( gblayout );

```

下面的语句将创建一个 GridBagLayout 对象：

```

gbConstraints = new GridBagConstraints ( )

```

上述语句之后的语句创建了各个组件。

文本区域是添加到 GridBagLayout 中的第一个组件。因为没有设置 weightx 和 weighty 的值，所以默认为 0。这样，即使有空间，组件的大小也不会发生变化。不过，由于文本区域占用了多行，所以它在垂直方向的长度受 b2 和 b3 的 weightx 值影响。当 b2 或 b3 在垂直方向上改变大小时，文本区域也会随之而改变。fill 变量的值已设置为 BOTH，这样文本区域将总是占满整个区域。这里没有指定 anchor 的值，所以就用它的默认值 CENTER。本程序中没有使用 anchor 变量，因此所有的组件都将默认地使用 CENTER 值。

下面的语句调用了程序员定义的方法 addComponent：

```
addComponent (ta, gbLayout , gbConstraints , 0, 0, 1, 3 );
```

前3个参数分别是TextArea对象、GridBagLayout对象和GridBagConstraints对象,后4个参数分别表示行坐标、列坐标、所占的列数和行数。

在addComponent方法中,声明了一个Component对象c、一个GridBagConstraints对象g、一个GridBagConstraints对象gc,以及四个整型变量row、column、width和height。下面的语句用于设置GridBagConstraints变量gridx和gridy的值:

```
gc.gridx = column;  
gc.gridy = row;
```

gridx变量设置为列坐标的值,gridy变量设置为行坐标的值。下面的语句用于设置GridBagConstraints变量gridwidth和gridheight的值:

```
gc.gridwidth = width;  
gc.gridheight = height;
```

gridwidth变量的值设置为列数,gridheight变量的值设置为行数。下面的语句用于设置GridBagConstraints:

```
g.setConstraints ( c, gc );
```

GridBagLayout的setConstraints方法带有两个参数:Component和GridBagConstraints,add方法用于将组件添加到applet中。

接下来添加的组件是按钮对象b1。weightx和weighty的值仍然是0。fill变量的值设置为HORIZONTAL,即该组件将在水平方向占满组件区,而垂直方向不占满。由于weighty的值为0,所以该按钮的高度永远不会发生变化。applet的大小在垂直方向上发生变化,这不会影响b1。按钮b1所处的位置是第0行、第1列,并且占有1行2列。

接下来添加的组件是选择按钮cb。weightx和weighty的值为0,fill变量的值为HORIZONTAL,因此该选择按钮只能在水平方向伸展。注意,这里没有重新设置weightx、weighty和fill变量的值。选择按钮cb所处的位置是第2行、第1列,并且占有2行1列。

按钮b2的weightx的值设置为1000,weighty的值设置为1。这样,该按钮所占有的区域在水平方向和垂直方向上都可以伸展。fill变量设置为BOTH,表示该按钮总是占满整个组件区。当applet的大小发生改变时,b2也将随之发生变化。按钮b2所处的位置是第1行、第1列,并且占有1行1列。

接下来添加按钮b3。weightx和weighty的值都设置为0,fill的值为BOTH。如果applet的大小发生改变,那么按钮b3也将随之发生变化;这是受b2的weight值的影响。注意,b2的weightx值比b3的weightx值大得多,因此当组件的大小发生变化时,b2将占用更大的区域。按钮b3所处的位置是第1行、第2列,并且占有1行1列。

文本字段和画板的weightx值和weighty值都为0,fill值也为BOTH。文本字段所处的位置为第3行、第0列,画板所处的位置为第3行、第2列。文本字段占有1行2列,画板占有1行1列。

Java还提供了另一种GridBagLayout。其中,gridx、gridy、gridwidth和gridheight变量都没有使用,只使用了用于确定位置的常量。GridBagConstraints常量是RELATIVE和REMAINDER,RELATIVE表示将组件放到上一个组件的后面,REMAINDER表示将组件放到当前行的最后。图11.27就是使用这些常量来排放GridBagLayout中的组件。该程序仅创建了GUI,没有其他功能。



```
1 // Fig. 11.27: GridBagDemo2.java
2 // Demonstrating GridBagLayout constants.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class GridBagDemo2 extends Applet {
7     private Choice cb;
8     private TextField tf;
9     private List m;
10    private Button b1, b2, b3, b4;
11    private GridBagLayout gblayout;
12    private GridBagConstraints gbConstraints;
13
14    public void init()
15    {
16        gblayout = new GridBagLayout();
17        setLayout( gblayout ); // applet
18
19        // instantiate gridbag constraints
20        gbConstraints = new GridBagConstraints();
21
22        // create some components
23        cb = new Choice( );
24        cb.addItem( "Pine" );
25        cb.addItem( "Ash" );
26        cb.addItem( "Pecan" );
27        tf = new TextField( "TextField" );
28        m = new List( 3, false );
29        m.addItem( "Java" );
30        b1 = new Button( "eins" );
31        b2 = new Button( "zwei" );
32        b3 = new Button( "drei" );
33        b4 = new Button( "vier" );
34
35        // textfield
36        gbConstraints.weightx = 1;
37        gbConstraints.weighty = 1;
38        gbConstraints.fill = GridBagConstraints.BOTH;
39        gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
40        addComponent( tf, gblayout, gbConstraints );
41
42        // button b1
43        // weightx and weighty are 1: fill is BOTH
44        gbConstraints.gridwidth = GridBagConstraints.RELATIVE;
45        addComponent( b1, gblayout, gbConstraints );
46
47        // button b2
48        // weightx and weighty are 1: fill is BOTH
49        gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
50        addComponent( b2, gblayout, gbConstraints );
51
52        // choicebox
53        // weightx is 1: fill is BOTH
54        gbConstraints.weighty = 0;
55        gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
```

```

56      addComponent( cb, gbLayout, gbConstraints );
57
58      // button b3
59      // weightx is 1: fill is BOTH
60      gbConstraints.weighty = 1;
61      gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
62      addComponent( b3, gbLayout, gbConstraints );
63
64      // button b4
65      // weightx and weighty are 1: fill is BOTH
66      gbConstraints.gridwidth = GridBagConstraints.RELATIVE;
67      addComponent( b4, gbLayout, gbConstraints );
68
69      // list
70      // weightx and weighty are 1: fill is BOTH
71      gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
72      addComponent( m, gbLayout, gbConstraints );
73  }
74
75  // addComponent is programmer-defined
76  private void addComponent( Component c, GridBagLayout g,
77                           GridBagConstraints gc )
78  {
79      g.setConstraints( c, gc );
80      add( c );      // add component to applet
81  }
82  }

```

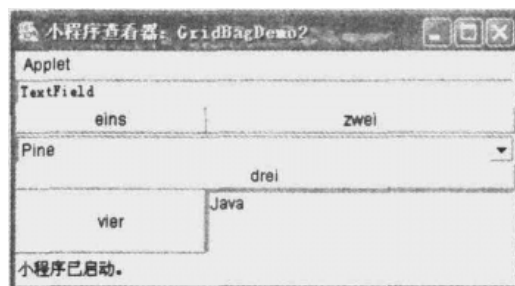


图 11.27 GridConstraints 常量 RELATIVE 和 REMAINDER

下面的语句创建了一个 GridBagLayout 对象，并将布局管理器设置成 GridBagLayout：

```

gbLayout = new GridBagLayout ( ) ;
setLayout ( gbLayout );

```

这里分别创建了放置在 GridBagLayout 中的各组件，即四个按钮、一个文本字段、一个列表及一个选择按钮。

首先添加文本字段。weightx 和 weighty 设置为 1，fill 变量设置为 BOTH。下面的语句将文本字段设置为一行的最后一个组件：

```
gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
```

通过调用程序员自定义的 addComponent 方法，将该文本字段添加到 applet 上。

addComponent 方法带有 Component 参数、GridBagLayout 参数和 GridBagConstraints 参数。首先使

用 GridBagLayout 的 setConstraints 方法设置组件, 然后利用 add 方法将组件添加到 applet 上。

按钮 b1 的 weightx 和 weighty 的值为 1, fill 变量的值为 BOTH, 下面的语句将按钮 b1 放在前一个组件的后面:

```
gcConstraints.gridwidth = GridBagConstraints.RELATIVE;
```

然后调用 addComponent 方法将该按钮添加到 applet 上。

按钮 b2 的 weightx 和 weighty 值为 1, fill 变量的值为 BOTH。该按钮是这一行的最后一个组件, 所以使用 REMAINDER 常量。最后调用 addComponent 方法将该按钮添加到 applet 中。

选择按钮的 weightx 值为 1, weighty 值为 0, 因此该选择按钮在垂直方向上将不能伸展。它是本行的惟一组件, 所以使用 REMAINDER 常量。最后调用 addComponent 方法将该选择按钮添加到 applet 上。

按钮 b4 的 weightx 和 weighty 值为 1, fill 变量的值为 BOTH。该按钮不是这一行的最后一个组件, 所以使用 REMAINDER 常量。最后调用 addComponent 方法将该按钮添加到 applet 中。

列表组件的 weightx 和 weighty 值为 1, fill 变量的值为 BOTH。该列表是这一行的最后一个组件, 所以使用 REMAINDER 常量。最后调用 addComponent 方法将该列表添加到 applet 中。

## 11.12 不使用布局管理器

Java 允许程序员不使用任何布局管理器。如果没有使用布局管理器, 那么每个独立组件的位置都必须由程序员手工安排。图 11.28 的程序没有使用布局管理器, 其中包括三个按钮和一个选择按钮。该程序仅创建了一个 GUI, 没有其他功能。

```
1 // Fig. 11.28: NoLayout.java
2 // Demonstrating the null layout manager.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class NoLayout extends Applet {
7     private Choice cb;
8     private Button b1, b2, b3;
9
10    public void init()
11    {
12        // do not use a layout manager
13        setLayout( null );
14
15        cb = new Choice();
16        cb.addItem( "Item 1" );
17        cb.addItem( "Item 2" );
18        cb.addItem( "Item 3" );
19
20        b1 = new Button( "Button" );
21        b2 = new Button( "Another Button" );
22        b3 = new Button( "Last Button" );
23
24        add( cb );
25        add( b1 );
26        add( b2 );
```

```

27         add( b3 );
28
29         b1.reshape( 15, 9, 60, 26 );
30         b2.reshape( 100, 40, 90, 22 );
31         b3.reshape( 220, 20, 70, 55);
32         cb.reshape( 50, 80, 70, 17);
33     }
34 }

```

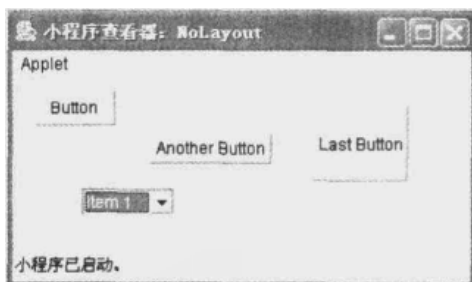


图 11.28 未使用布局管理器的程序

NoLayout 声明了一个选择按钮和三个按钮，接着下面的语句表示不使用任何布局管理器：

```
setLayout ( null );
```

然后创建选择按钮对象cb，并为它添加三个选项。接着创建各个按钮对象，并添加到applet中。下面的语句用于设置各组件的位置和大小：

```

b1.reshape ( 15, 9, 60, 26 );
b2.reshape ( 100 , 40, 90, 22 );
b3.reshape ( 220, 20, 70, 55 );
cb.reshape ( 50, 80, 70, 17 );

```

Component类的reshape方法用于确定组件的x坐标、y坐标、宽度和高度。

#### 常见编程错误 11.17

把组件放到容器边界的外部会产生逻辑错误，该组件将会被剪切或看不见。

#### 常见编程错误 11.18

忘记将组件手工放置到容器上会产生逻辑错误，该组件通常会不可见。

#### 可移植性技巧 11.1

如果不使用布局管理器（即手工放置组件），可能会影响程序运行平台的独立性。

## 11.13 程序员自定义的布局管理器

如果前面提供的任何一种布局管理器都无法满足用户对组件布局的要求，那么Java可以让用户创建自己的布局管理器。程序员自定义的布局管理器是通过LayoutManager接口创建的。

#### 编程技巧 11.5

创建程序员自定义的布局管理器会使程序变得复杂，所以应尽可能使用系统预定义的布局管理器。

我们编写一个程序，利用自定义的布局管理器将5个组件排列成“V”字形，如图11.29所示。

这个例子充分说明了如何创建一个程序员自定义的布局管理器,并为创建更复杂的布局管理器打下了基础。

#### 软件工程视点 11.4

创建程序员自定义布局管理器的功能是 Java 的另一种扩展功能。

UserLayout 类创建了 5 个组件,并使用下面的语句将布局管理器设置为 VLayout:

```
setLayout (new VLayout ( ));
```

然后将各组件添加到 applet 上。构造函数 VLayout 用于创建布局管理器。

VLayout 类的 LayoutManager 接口提供了 5 个方法,这些方法必须重写。这 5 个方法是 addLayoutComponent、layoutContainer、minimumLayoutSize、preferredLayoutSize 和 removeLayoutComponent。addLayoutComponent 方法的功能是确定如何向布局中添加指定的组件,layoutContainer 方法的功能是确定如何布局容器,minimumLayoutSize 方法的功能是确定布局所需的最小值,preferredLayoutSize 方法的功能是确定容器布局的最佳尺寸,removeLayoutComponent 方法的功能是从布局中删除一个指定的组件。这里我们只为 LayoutContainer 方法编写了实际的功能。

---

```
1 // Fig. 11.29: UserLayout.java
2 // Demonstrating a programmer-defined layout manager.
3 import java.applet.Applet;
4 import java.awt.*;
5
6 public class UserLayout extends Applet {
7     private Choice cb1, cb2;
8     private Button b1, b2, b3;
9
10    public void init()
11    {
12        // use the v layout manager
13        setLayout( new VLayout() );
14
15        cb1 = new Choice();
16        cb1.addItem( "Item 1" );
17        cb1.addItem( "Item 2" );
18
19        cb2 = new Choice();
20        cb2.addItem( "Choice 1" );
21        cb2.addItem( "Choice 2" );
22
23        b1 = new Button( "Button 1" );
24        b2 = new Button( "Button 2" );
25        b3 = new Button( "Button 3" );
26
27        // order is important
28        add( cb1 );
29        add( b1 );
30        add( b2 );
31        add( b3 );
32        add( cb2 );
33    }
34 }
35
```

```
36 // create class for user defined layout
37 class VLayout implements LayoutManager {
38
39     public void layoutContainer( Container c )
40     {
41         int numberOfComponents = c.countComponents();
42
43         if ( numberOfComponents > 5 )
44             numberOfComponents = 5;
45
46         Insets i = c.insets();
47         int w = c.size().width - i.left - i.right;
48         int h = c.size().height - i.bottom - i.top;
49         int x = 0, y = 0;
50
51         // calculate x and y values for each component
52         for ( int j = 0; j < numberOfComponents; j++ ) {
53             Component comp = c.getComponent( j );
54             Dimension d = comp.preferredSize();
55
56             switch ( j ) {
57                 case 0: // first component
58                     x = ( int ) ( 0.2 * w );
59                     y = ( int ) ( 0.25 * h );
60                     break;
61                 case 1: // second component
62                     x = ( int ) ( 0.3 * w );
63                     y = ( int ) ( 0.5 * h );
64                     break;
65                 case 2: // third component
66                     x = ( int ) ( 0.4 * w );
67                     y = ( int ) ( 0.75 * h );
68                     break;
69                 case 3: // fourth component
70                     x = ( int ) ( 0.5 * w );
71                     y = ( int ) ( 0.5 * h );
72                     break;
73                 case 4: // fifth component
74                     x = ( int ) ( 0.6 * w );
75                     y = ( int ) ( 0.25 * h );
76                     break;
77             }
78
79             // size the component
80             comp.reshape( x, y, d.width, d.height );
81         }
82     }
83
84     // These last four methods must be overridden
85     // However our layout manager does not use them
86     public void addLayoutComponent( String s, Component c )
87     { } // empty
88
89     public void removeLayoutComponent( Component c )
90     { } // empty
91
92     public Dimension preferredLayoutSize( Container c )
```

```

93     { return minimumLayoutSize( c ); }
94
95     public Dimension minimumLayoutSize( Container c )
96     { return new Dimension( 0, 0 ); }
97 }

```



图 11.29 一个程序员自定义的布局管理器

`LayoutContainer` 方法带有一个参数,即要进行布局的容器。下面的语句使用了 `Container` 类的 `countComponents` 方法来求出容器中组件的数目:

```
int numberOfComponents = c.countComponents();
```

如果组件的数目超过5,那么 `numberOfComponents` 变量的值就设置成5。下面的语句是用 `Container` 类的 `insets` 方法求出容器的 `inset` 对象:

```
Insets i = c.insets();
```

该 `Insets` 对象中保存的是容器4个边界的宽度值。下面的语句用于计算容器的中间区域(去掉边界之后的区域)的宽度和高度:

```
int w = c.size().width - i.left - i.right;
int h = c.size().height - i.bottom - i.top;
```

`w` 和 `h` 用于准确地定位组件。`Component` 类的 `size` 方法返回一个 `Dimension` 对象, `Dimension` 对象中包含两个实例变量 `width` 和 `height`。`Inset` 类中包含4个实例变量——`left`、`right`、`top` 和 `bottom`, 这些变量分别对应于左边界、右边界、上边界和下边界的宽度值。

在 `for` 循环中,每循环一次就处理一个组件。下面的语句将根据添加的顺序来获取组件:

```
Component comp = c.getComponent( j );
```

`Container` 类的 `getComponent` 方法用于返回指定的组件。下面的语句创建了 `Dimension` 对象 `d`:

```
Dimension d = comp.preferredSize();
```

并将 `preferredSize` 方法返回的值赋给 `d`。注意,我们使用 `preferredSize` 方法仅仅是返回了一个 `Dimension` 对象,它的宽度和高度都为0。

#### 常见编程错误 11.19

如果 `getComponent` 方法的输入值大于组件的实际数目,将会引发 `ArrayIndexOutOfBoundsException` 异常。

在 switch 结构中, 分别设置了每个组件的 x 坐标和 y 坐标的位置。第一个组件位于 20% 宽度和 25% 高度处, 第二个组件位于 30% 宽度和 50% 高度处, 第三个组件位于 40% 宽度和 75% 高度处, 第四个组件位于 50% 宽度和 50% 高度处, 第五个组件位于 60% 宽度和 25% 高度处, 每个组件都利用 reshape 方法定位和设置大小。

## 小结

- 文本区域是一个可以显示多行文本的区域, 其中总是带有一个垂直滚动条和一个水平滚动条, 仅当文本区域中的文本超出文本区域的可见范围时, 滚动条才能滚动。需要有一个外部事件来触发从文本区域中获取文本的操作, 比如按下 一个按钮。
- TextComponent 类的 setText 方法用于设置文本区域中的文本, getSelectedText 方法用于返回文本区域中选中的文本。
- 画板是一个可以画图 and 接收鼠标事件的组件, Canvas 类是从 Component 类继承的, 画板有它自己的图形环境。
- Component 的 resize 方法用于设置组件的大小。
- 如果 在画板上画图, 必须重写 paint 方法。画板的坐标都是相对于画板的左上角(0,0)。
- 滚动条是一个可以在某个整数范围内“滚动”的组件。
- 滚动条的方向由常量 Scrollbar.HORIZONTAL 或 Scrollbar.VERTICAL 确定。当使用鼠标点击滚动箭头、滚动块或是滚动箭头和滚动块之间的区域时, 将会产生 Scrollbar 事件, 该事件由 handleEvent 方法处理。
- 定制组件是通过扩展 Canvas 类或 Panel 类来创建的, 包含其他组件的定制组件应扩展 Panel 类。
- 框架是一个带有标题栏和边界的窗口。Frame 类是 Window 类的扩展类, Window 类中包含一些用于窗口管理的方法。窗口和框架的默认布局管理器是 BorderLayout。
- Frame 的 dispose 方法用于释放框架所占用的资源, 在显示框架之前必须设置其大小, 默认情况下框架是不可见的。程序员必须显式地调用 show 来显示框架。
- Frame 的 setResizable 方法用于设置是否允许用户调整框架大小, 默认情况下框架的大小是可调整的。
- 当点击框架的退出图标时, 就会产生 WINDOW\_DESTROY 事件。
- 利用菜单, 用户可以完成一些直接或间接的动作, 而不必添加额外的 GUI 组件使容器“拥挤不堪”。菜单只能用在框架上。
- MenuComponent 类是 MenuBar 和 MenuItem 的基类。
- MenuBar 类中包含一些用于管理菜单栏的构造函数和方法。当使用鼠标点击一个菜单时, 该菜单就会展开, 并显示一系列菜单项。点击一个菜单项会产生一个动作事件。
- CheckoxMenuItem 是一个带有选择标记的菜单项。当选中复选框选项时, 在该菜单项名称的左边就会出现一个选择标记。如果再次选中该复选框选项, 那么它左边的选择标记就会消失。
- Menu 类的 add 方法用于将一个菜单项添加到菜单中, MenuBar 类的 add 方法用于将一个菜单添加到菜单栏中, Frame 类的 setMenuBar 方法用于将菜单栏设置到框架上。
- 分隔条是一条能将菜单项显式分开的线。将 MenuItem 构造函数的参数设置为连字符“-”, 就可以创建一个分隔条。



- `CheckboxMenuItem` 和 `setState` 方法用于设置复选框菜单项的布尔状态, `getState` 方法用于获取复选框菜单项当前的布尔状态。
- 对话框是一个带有标题栏的无边界窗口。对话框通常用于接收用户的信息或向用户显示信息。对话框由 `Dialog` 类创建, `Dialog` 类是从 `Window` 类继承的。对话框分为模态的和非模态的两种。当模态的对话框打开时, 不允许访问应用程序中的其他窗口, 直到该对话框关闭。而当非模态的对话框打开时, 用户仍然可以访问其他窗口。对话框的默认布局管理器是 `BorderLayout`。
- `MenuBar` 的 `setHelpMenu` 方法用于将 help 菜单添加到菜单栏上。
- `MenuItem` 的 `enable` 方法和 `disable` 方法分别用于将菜单项设置为激活的和未激活的。如果一个菜单项为激活的, 那么选择该菜单项会产生一个动作事件。如果菜单项为未激活的, 那么它就会显示为灰色——选中显示为灰色的菜单不会产生任何事件。
- 文件对话框默认为模态的, 一共有两种文件对话框——Open 文件对话框和 Save 文件对话框。
- `CardLayout` 布局管理器是将各组件排放成一组卡片的形式, 只有最上面的卡片可见。每张卡片通常都是一个容器 (例如面板)。在卡片上可以使用任何布局管理器。`CardLayout` 的 `first`、`previous`、`next` 和 `last` 方法用于显示卡片。
- `GridBagLayout` 布局管理器与 `GridLayout` 布局管理器类似, 因为 `GridBagLayout` 也是将组件放置在网格中。不过 `GridBagLayout` 更灵活一些, 各组件的大小可以不一样, 并且可以按任何顺序添加。`GridBagConstraints` 用于确定如何使用 `GridBagLayout` 来设置组件布局。
- Java 允许程序员不使用任何布局管理器, 只需将 `setLayout` 方法的参数设置为 `null` 即可。如果没有使用布局管理器, 那么每个独立组件的位置和大小都必须由程序员手工安排。
- `Component` 类的 `reshape` 方法用于确定组件的 `x` 坐标、`y` 坐标、宽度和高度。
- 程序员自定义的布局管理器是通过 `LayoutManager` 接口创建的, `LayoutManager` 接口中包含 5 个方法, 必须重写这些方法, 它们分别是 `addLayoutComponent`、`layoutContainer`、`minimumLayoutSize`、`preferredLayoutSize` 和 `removeLayoutComponent`。
- `Container` 类的 `countComponents` 方法用于获取容器中的组件数目, `insets` 方法用于获取容器的 `Insets` 对象。
- `Component` 类的 `size` 方法将返回一个 `Dimension` 对象, 其中包含两个实例变量 `width` 和 `height`。

## 术语

action method    action 方法

ACTION\_EVENT

add method    add 方法

addLayoutContainer method    addLayoutContainer  
方法

anchor

arrow head symbol    箭头符号

bottom    底部

bottom border    下边界

canvas    画板

Canvas class    Canvas 类

Canvas constructor    Canvas 构造函数

CardLayout class    CardLayout 类

CardLayout layout manager    CardLayout 布局管理器

CardLayout object    CardLayout 对象

CardLayout reference    CardLayout 引用

checkbox menu item    复选框菜单项

CheckboxMenuItem class    CheckboxMenuItem 类

CheckboxMenuItem constructor

CheckboxMenuItem 构造函数

- clipping 剪切
- Component class Component 类
- Component reference Component 引用
- composite component 组合组件
- Container class Container 类
- countComponents method countComponents 方法
- custom component 定制组件
- dialog box 对话框
- Dialog class Dialog 类
- Dialog constructor Dialog 构造函数
- Dimension class Dimension 类
- disable method disable 方法
- dispose method dispose 方法
- enable method enable 方法
- FileDialog class FileDialog 类
- FileDialog.SAVE
- FileDialog.LOAD
- fill
- first method first 方法
- frame 框架
- frame-based application 基于框架的应用程序
- Frame class Frame 类
- Frame constructor Frame 构造函数
- Frame object Frame 对象
- getComponent method getComponent 方法
- getSelectedText method getSelectedText 方法
- getValue method getValue 方法
- graphical user interface 图形用户界面
- grayed 灰度显示
- GridBagConstraints class GridBagConstraints 类
- GridBagConstraints constructor GridBagConstraints 构造函数
- GridBagConstraints reference GridBagConstraints 引用
- GridBagConstraints.CENTER
- GridBagConstraints.EAST
- GridBagConstraints.NORTH
- GridBagConstraints.NORTHEAST
- GridBagConstraints.NORTHWEST
- GridBagConstraints.RELATIVE
- GridBagConstraints.REMAINDER
- GridBagConstraints.SOUTH
- GridBagConstraints.SOUTHEAST
- GridBagConstraints.SOUTHWEST
- GridBagConstraints.WEST
- GridBagLayout constructor GridBagLayout 构造函数
- GridBagLayout layout manager GridBagLayout 布局管理器
- GridBagLayout reference GridBagLayout 引用
- gridheight
- gridwidth
- gridx
- gridy
- handleEvent method handleEvent 方法
- height
- hide
- inset
- Inset class Inset 类
- inset method inset 方法
- Inset object Inset 对象
- last method last 方法
- layoutContainer method layoutContainer 方法
- LayoutManager interface LayoutManager 接口
- left
- left border 左边界
- maximum value 最大值
- menu 菜单
- menu bar 菜单栏
- menu items 菜单项
- Menu class Menu 类
- Menu constructor Menu 构造函数
- menu bar name 菜单栏名称
- menu item name 菜单项名称
- menu name 菜单名
- MenuBar class MenuBar 类
- MenuBar constructor MenuBar 构造函数
- MenuComponent class MenuComponent 类
- MenuItem class MenuItem 类
- MenuItem constructor MenuItem 构造函数
- minimum value 最小值
- minimumLayoutSize method minimumLayoutSize 方法

- model 模态的
- modeless 非模态的
- next method next 方法
- open file dialog 打开文件对话框
- preferredSize method preferredSize 方法
- previous method previous 方法
- programmer-defined layout manager 自定义的布局管理器
- removeLayoutComponent method  
removeLayoutComponent 方法
- reshape method reshape 方法
- resize method resize 方法
- right border 右边界
- save file dialog 保存文件对话框
- scroll arrow 滚动箭头
- Scrollbar class Scrollbar 类
- Scrollbar constructor Scrollbar 构造函数
- scrollbar events 滚动条事件
- Scrollbar.HORIZONTAL
- scrollbar orientation 滚动条的方向
- Scrollbar reference Scrollbar 引用
- Scrollbar.VERTICAL
- scroll box 滚动块
- separator bar 分隔条
- setBackground method setBackground 方法
- setConstraints method setConstraints 方法
- setEditable method setEditable 方法
- setFont method setFont 方法
- setHelpMenu method setHelpMenu 方法
- setLayout(null);
- setMenuBar constructor setMenuBar 构造函数
- setResizable method setResizable 方法
- setState method setState 方法
- setText method setText 方法
- show method show 方法
- size method size 方法
- slider 滑块
- submenu 子菜单
- super
- text area 文本区域
- TextArea class TextArea 类
- TextArea constructor TextArea 构造函数
- TextArea object TextArea 对象
- TextArea reference TextArea 引用
- TextComponent class TextComponent 类
- title bar 标题栏
- top
- top border 上边界
- weightx
- weighty
- width 宽度
- Window class Window 类
- Window object Window 对象
- WINDOW\_DESTROY
- WINDOW\_DESTROY events
- WINDOW\_DESTROY 事件

## 自测练习

### 11.1 填空:

- a) \_\_\_\_\_ 类用于创建菜单对象。
- b) 如果要创建分隔条, 必须使用 \_\_\_\_\_ 构造函数。
- c) 将文本区域的 \_\_\_\_\_ 方法的参数设置为 true, 就使文本区域中的文本可编辑。
- d) 滚动条事件由 \_\_\_\_\_ 方法处理。
- e) 对于菜单项来说, Event 类的实例变量 arg 的值是 \_\_\_\_\_。
- f) GridBagConstraints 的实例变量 \_\_\_\_\_ 的默认值为 CENTER。

### 11.2 判断下列句子是否正确。如果不正确, 请解释原因。

- a) 当创建对话框时, 必须至少创建一个菜单, 并将其添加到对话框中。
- b) 当创建框架时, 必须至少创建一个菜单, 并将其添加到框架中。

- c) 静态变量 fill 属于 GridBagLayout 类。
  - d) 框架和 applet 不能同时在一个程序中使用。
  - e) 框架或 applet 的左上角坐标为(0,0)。
  - f) 文本区域中的文本总是静态的 (只读的)。
- 11.3 找出下列语句中的错误, 并进行改正。
- a) `if( x instanceof TextArea )// check for text area event`
  - b) `Menubar b;//create menubar reference`
  - c) `myScroll = Scrollbar ( 1000, 0, 222, 100, 450 );`
  - d) `gbc.fill = GridBagConstraints.NORTHWEST ; // set fill`
  - e) `m1.add ( new Checkbox ( " Snap to grid " ) );// m1 is a Menu`
  - f) `// call Java 's HelpMenu constructor to create a help Menu`  
`help = new HelpMenu ( " Help " );`
  - g) `y.reshape(x, y, w, h, c ); // size and place component`

## 自测练习答案

- 11.1 a) Menu。b) MenuItem。c) setEditable。d) handleEvent。e) 选中的菜单项 (字符串)。  
 f) anchor。
- 11.2 a) 不正确。对话框可以不包含菜单。  
 b) 不正确。框架可以不包含菜单。  
 c) 不正确。变量 fill 属于 GridBagConstraints 类。  
 d) 不正确。它们可以同时使用。  
 e) 正确。  
 f) 不正确。默认情况下文本区域是可编辑的。
- 11.3 a) 文本区域本身不会产生事件, 必须使用一个外部事件, 比如按钮事件。  
 b) Menubar 应改为 MenuBar。  
 c) 第一个参数的值应是 Scrollbar.HORIZONTAL 或 Scrollbar.VERTICAL。  
 d) 常量值应为 BOTH、HORIZONTAL、VERTICAL 或 NONE。  
 e) 复选框不能添加到菜单中, 应使用 CheckboxMenuItem。  
 f) Java 中没有 HelpMenu 类构造函数, 应该使用 setHelpMenu 方法来设置 help 菜单。  
 g) reshape 方法应该有 4 个参数, 而不是 5 个。

## 练习

- 11.4 填空:
- a) \_\_\_\_\_ 是通过继承 Canvas 类而创建的。
  - b) 包含一个菜单的菜单项称为 \_\_\_\_\_。
  - c) 文本字段和文本区域都是直接从 \_\_\_\_\_ 类继承的。
  - d) 构造函数 \_\_\_\_\_ 用于显示菜单栏。
  - e) 为了创建分隔条, 必须将 MenuItem 构造函数的参数设置为 \_\_\_\_\_。

- f) size 方法属于 \_\_\_\_\_ 类
- g) 如果要创建一个程序员自定义的布局管理器, 必须提供 \_\_\_\_\_ 接口
- h) \_\_\_\_\_ 类是从 Dialog 类继承的。
- 11.5 判断下列句子是否正确。如果不正确, 请解释原因。
- 使用菜单必须要创建一个 MenuBar 对象。
  - Canvas 对象能够处理鼠标事件。
  - CardLayout 是框架的默认布局管理器。
  - setEditable 方法是一个 TextComponent 类方法。
  - GridBagLayout 布局管理器提供了 LayoutManager。
  - Canvas 对象上不能使用任何布局管理器。
  - Frame 类是直接来自 Window 类继承的。
  - 文件对话框默认为模态对话框。
- 11.6 找出下列语句中的错误, 并进行改正。
- `x.add( new MenuItem( " Submenu Color " ) ); // Create submenu`
  - `setLayout( m = new GridbagLayout ( ) );`
  - `if( u instanceof Canvas )`  
`System.out.println( " It was a canvas ! " );`
  - `int weightValue = i.weightx ; // i is an Inset Object`
  - `String s = TextArea.getText ( ) ;`
  - `Dialog d = new Dialog ( a , " Dialog " , true ); // a is an applet`
  - `int width = z.width( ) ; // z is a Dimension object`
- 11.7 编写一个程序, 随机绘制一个圆形, 然后计算并显示该圆形的面积、半径、直径和周长。利用下面的公式进行计算:
- $$\begin{aligned} \text{直径} &= 2 \times \text{半径} \\ \text{面积} &= \pi \times \text{半径}^2 \\ \text{周长} &= 2 \times \pi \times \text{半径} \end{aligned}$$
- 其中,  $\text{Pi}(\pi)$  的值使用常量 `Math.PI` 表示。所有的图形都画在画板上, 计算结果则显示在一个只读文本区域中。
- 11.8 增加练习 11.7 中程序的功能, 允许用户利用滚动条来选择半径的值。在本程序中, 半径的范围限制在 100 到 200 之间。当半径发生变化时, 直径、面积和周长的值也随之变化, 并将其显示出来。半径的初始值设为 150, 利用练习 11.7 中给出的公式进行计算。仍将所有的图形画在画板上, 在一个只读文本区域中显示计算结果。
- 11.9 修改图 11.26 的程序, 改变 `weightx` 和 `weighty` 的值, 以观察所产生的影响。当一个组件具有非 0 的 `weight` 值, 但却不允许它填满整个区域时 (即 `fill` 的值不是 `BOTH`), 将会产生什么结果?
- 11.10 将图 11.29 的程序作为模板, 编写一个自定义的布局管理器。如果需要其他信息, 可以参阅关于 Java API 的部分。
- 11.11 编写一个程序, 利用 `paint` 方法显示出画板滚动条的当前值。另外, 提供一个文本字段作为输出区域, 随时显示滚动条的当前值。再提供一个标签, 用于标识该文本字段, 这里使用 `Scrollbar` 类的 `setValue` 方法和 `getValue` 方法。注意: `setValue` 方法是一个 `public`

方法，没有返回值，只带有一个整型参数，即滚动条的值。

- 11.12 修改图 11.22 的程序，使用一个 Choice 按钮代替四个单独的按钮，不要修改“卡片”。
- 11.13 修改图 11.22 的程序，向“卡片”盒中至少再添加两张新“卡片”。
- 11.14 编写一个程序，让用户查看某一指定的颜色。该程序应包含三个只读文本字段、一个画板及三个滚动条。画板用于显示颜色，滚动条分别用于调整红色、绿色和蓝色的值，三个文本字段分别用于显示相应的滚动条的当前值。
- 11.15 创建一个定制组件。如果可能，使定制组件处理它自己产生的事件。
- 11.16 编写一个程序，使用其他的 TextArea 类构造函数。这些构造函数与本章中介绍的两个构造函数有什么区别？

# 第12章 异常处理

## 教学目标

- 理解异常和错误处理
- 学会使用 try 程序块来描述可能产生异常的代码
- 学会抛出异常
- 学会使用 catch 程序块来进行异常处理
- 学会使用 finally 程序块来释放资源
- 理解 Java 异常的继承层次结构
- 学会创建自定义异常

## 12.1 简介

在这一章中,我们将介绍有关异常处理的详细内容。Java的功能扩展增加了可能发生错误的数目和类型,每一个新增加的类中都可能包含一些错误。本章介绍的这些技术将有助于程序员编写出更清晰、更强大、容错性更好的程序。另外,本章还将说明不应使用异常处理的情况。

在本章中,对Java异常处理的介绍部分基于Andrew Koenig和Bjarne Stroustrup的论文《C++的异常处理(修订版)》(“Exception Handling for C++ (revised)”)。这篇文章发表于1990年4月在旧金山举行的“Proceedings of the USENIX C++ Conference”,它奠定了C++异常处理的基础。Java的设计者们选用了与C++类似的异常处理机制。

根据应用程序的软件系统,以及所开发的软件是否为要发行的产品,应用程序中的错误处理程序在内容和数量上都有所不同。和一般软件不同,用于“发行”的软件产品趋向于采用更多的错误处理。

处理错误有很多种流行的方法。最一般的方法是将错误处理程序段分散到系统代码中,在可能发生错误的位置上处理错误。这种方法的优点是程序员在阅读程序时可以立刻在该程序的附近看到错误处理部分,进而确定所做的处理是否合适。

采用这种方法的缺点是,从某种定义上来看,错误处理程序“搅乱”了程序代码。程序员很难直接关注应用程序本身,查看其是否正确地完成了预定的功能,这样就使理解和维护应用程序变得很困难。

一些常见的异常实例包括:数组下标越界、算法溢出(即超出了数值的表达范围)、除数为零、无效参数及内存溢出等。

### 编程技巧 12.1

使用Java的异常处理功能可以将错误处理程序与程序执行的“主线”分离开,以便提高程序的清晰度和可修改性。

异常处理功能使得应用程序可以主动地去捕获和处理错误,而不会任由错误出现并产生更大的

麻烦。异常处理功能主要是为处理一些同步错误而设计的,比如除数为零(当程序执行除法操作时可能会产生该错误)。异常处理功能不宜处理异步事件,比如磁盘 I/O 结束、网络信息到达、点击鼠标、敲击键盘等,这些事件最好通过其他的方法进行处理,比如 Java 事件监听。

可以在程序中使用 Java 异常处理来捕获各种类型的异常,捕获某一种类型的所有异常,或是捕获相关类型的所有异常。这一功能使程序可以提高在执行过程中处理问题的可能性,从而使程序的性能更加强大。

如果系统能够从故障状态恢复过来,则在这种情况下可以使用异常处理。这个恢复程序就称为异常处理程序,异常处理程序通常在导致异常的方法和调用方法中定义。

异常处理通常是在故障诊断和故障处理不在同一位置的情况下使用的。例如,如果程序必须一直和用户进行交互对话,那么就不应该使用异常处理来处理键盘输入时出现的问题。

#### 编程技巧 12.2

异常处理的位置与异常发生的位置必须不同。如果一个程序能够处理自己的错误,那么就利用传统的错误处理方法进行局部错误处理。

#### 编程技巧 12.3

要避免使用异常处理来代替错误处理,因为这样会降低程序的清晰性。

对于传统的程序控制,要避免使用异常处理技术,这里还有另一个原因。异常处理是为错误处理而设计的,而这种错误处理动作很少发生(通常会导致程序终止)。因此,不要指望使用异常处理来达到程序所期望的那种最佳性能。

#### 性能提示 12.1

尽管可以利用异常处理来代替错误处理,但是这会降低程序的性能。

#### 性能提示 12.2

当没有发生异常时,异常处理程序的存在对程序的执行很少或不会产生额外的影响。而当异常发生时,它就会占用额外的执行时间。

#### 测试与调试提示 12.1

异常处理有助于提高一个程序的容错性。

#### 编程技巧 12.4

让程序员使用 Java 的标准异常处理功能来代替他们各自的专用方法,可以在大型项目中提高程序的清晰度。

异常是从超类 `Exception` 继承的子类的对象。我们将讨论如何处理“未被捕获”的异常,以及 Java 如何处理无法预料的异常。我们将看到如何使用异常子类(从一个通用的异常子类派生而来)来表示相关类型的异常。

异常处理可以看成是从方法中返回或退出程序块的另一种方式。在正常情况下,当产生异常时,该异常将由产生该异常的方法的调用者来处理,或由该调用者的调用者来处理,或依次类推,由调用序列上的任意调用者进行处理。因此,必须在调用序列上找出该异常的处理程序。

#### 软件工程视点 12.1

异常处理尤其适用于那些分别开发的组件系统,这样的系统在现实世界的软件系统和产品中是很典型的。异常处理使各组件之间更容易组合,并且使它们能更有效地协作。



### 软件工程视点 12.2

程序员在使用其他不支持异常处理的语言进行编程时,经常会拖延错误处理程序的编写,有时甚至忘记了编写,这样就会降低程序的完整性以及软件产品的质量。Java强制程序员从项目一开始就着手进行异常处理。而且,程序员必须要投入很大的精力来将异常处理策略融合到软件产品中。

### 软件工程视点 12.3

最好在开始进行系统设计时就将异常处理策略融合到系统中,如果系统已经实现,那么就很难再添加有效的异常处理功能。

## 12.2 何时使用异常处理

异常处理应该用于下列情况:

- 当方法由于其无法控制的原因而不能实现其功能时;
- 处理来自某些程序组件的异常,在这些程序组件中不适合直接处理异常;
- 在大型项目中,对于每个项目都以一致的方式进行错误处理。

### 软件工程视点 12.4

类库的使用者对于类库中产生的每一个异常,可能都需要一个惟一的错误处理。要想在类库中实现能够满足所有使用者惟一需求的错误处理是不可能的。这时,异常处理就成为处理类库中错误的一种合适的方法。

## 12.3 其他的错误处理技术

在本章之前,我们已经介绍了几种处理异常情况的方法。第一种方法是程序中可以忽略异常。对于那些要向公众发行的软件产品来说,或用于处理关键任务的特殊软件来说,忽略异常无疑会导致重大的软件破坏。但是对于那些专为自己开发而不对外发布的软件来说,通常可以忽略很多种错误。第二种方法是在遇到异常情况时,程序可以根据指示终止运行。这就使程序不会带着错误继续运行直到结束,从而产生错误的运行结果。对于很多种错误来说,这不失为一个好方法。但是,这种策略对于那些处理关键任务的应用程序是不合适的。在这里,资源问题也很重要。如果为一个程序分配了一个资源,那么在它终止之前应正常返回该资源。

### 常见编程错误 12.1

终止一个程序可能会使某个资源永远占用而不能释放;这样,其他程序就无法申请该资源,因此可能会导致“资源泄漏”。

### 编程技巧 12.5

如果在一个方法中能够处理某一类型的异常,那么最好就在该方法中进行处理,而不要把它传递到其他部分去处理。这将使程序看起来更清晰。

### 性能提示 12.3

如果一个错误能够在本地处理,那么就直接处理,而不要抛出异常。这样可以提高程序的执行速度。异常处理比简单的本地处理的速度要慢。

## 12.4 Java 异常处理的基础

Java异常处理适用于在一个方法中能检测出错误但不能处理错误的情况。在这样的方法中将抛出一个异常。Java无法保证一定存在某个“内容”（即一个异常处理程序，当检测出一个异常时就执行该程序）能特别用于处理某种类型的异常。如果存在，那么将“捕获”和“处理”该异常。下面的“测试与调试提示”中描述了当没有找到合适的异常处理程序时会产生什么结果。

### 测试与调试提示 12.2

所有的Java applet和某些Java应用程序是基于GUI的。而另外一些Java应用程序则不是基于GUI的，这种程序通常称为命令行应用程序（或控制台应用程序）。在命令行应用程序中，当一个异常未被捕获时，该程序将在运行了默认的异常处理程序之后终止（即退出Java）。在applet或基于GUI的应用程序中，当一个异常未被捕获时，GUI在运行了默认的异常处理程序之后仍然会显示，而且用户可以继续使用该applet或应用程序。不过，此时的GUI可能处于一种不稳定的状态。

程序员将可能产生异常的代码包含在一个try程序块中。try程序块后面紧跟着一个或多个catch程序块。每个catch程序块都指定了它所能捕获的异常的类型，并包含一个相应的异常处理程序。在最后一个catch程序块的后面，可以提供一个可选的finally程序块，无论是否产生异常，该程序块都会执行。在本章后面我们将会看到，finally程序块是执行资源释放操作以防止“资源泄漏”的理想位置。

当抛出一个异常时，程序控制将离开try程序块，在各个catch程序块中按顺序查找适当的异常处理程序（后面将讨论哪一个是适当的处理程序）。如果抛出的异常的类型与某个catch程序块的参数类型相匹配，那么就执行该catch程序块中的代码。如果在一个try程序块中没有抛出任何异常，那么就跳过该程序块的异常处理程序，继续执行最后一个catch程序块之后的代码。如果在最后一个catch程序块之后有一个finally程序块，那么不论是否抛出异常，都要执行这个finally程序块。

我们可以使用throws子句来指定要抛出的异常。异常可以在方法的try程序块中抛出，也可以在try程序块中直接或间接调用的方法中抛出。执行throw语句的位置称为抛出点。

一旦抛出了一个异常，那么抛出该异常的程序块就会终止，程序控制将无法返回该抛出点。因此，Java所采用的是终止方式的异常处理，而不是恢复方式的异常处理。在恢复方式中，程序控制将返回抛出异常的抛出点，并恢复程序的执行。

当发生一个异常时，可以将异常附近的一些信息传递到异常处理程序中。这些信息包括抛出对象本身的类型，或从发生异常的位置附近收集并放入抛出对象中的信息。

### 软件工程视点 12.5

Java风格的异常处理的关键是，处理异常的程序或系统与产生异常的程序从形式上可以分开。

## 12.5 一个异常处理的简单实例：除数为零

现在让我们分析一个异常处理的简单实例。在图12.1的程序中，分别用try、throw和catch来检测、标识以及处理除数为零的异常。

```
1 // Fig. 12.1: DivideByZeroException.java
2 // Definition of class DivideByZeroException.
3 // Used to throw an exception when a
4 // divide-by-zero is attempted.
```

```
5      public class DivideByZeroException
6          extends ArithmeticException {
7          public DivideByZeroException()
8          {
9              super( "Attempted to divide by zero" );
10         }
11     }
12     // Fig. 12.1: DivideByZeroTest.java
13     // A simple exception handling example.
14     // Checking for a divide-by-zero-error.
15     import java.awt.*;
16     import java.applet.Applet;
17
18     public class DivideByZeroTest extends Applet {
19         Label prompt1, prompt2;
20         TextField input1, input2;
21         int number1, number2;
22         double result;
23
24         // Initialization
25         public void init()
26         {
27             prompt1 = new Label( "Enter numerator" );
28             input1 = new TextField( 10 );
29             prompt2 = new Label( "Enter denominator and " +
30                                 "press Enter" );
31             input2 = new TextField( 10 );
32             add( prompt1 );
33             add( input1 );
34             add( prompt2 );
35             add( input2 );
36         }
37
38         // Process GUI events
39         public boolean action( Event event, Object object )
40         {
41             if ( event.target == input2 ) {
42                 number1 = Integer.parseInt( input1.getText() );
43                 input1.setText( "" );
44                 number2 = Integer.parseInt( input2.getText() );
45                 input2.setText( "" );
46
47                 try {
48                     result = quotient( number1, number2 );
49                     showStatus( number1 + " / " + number2 + " = " +
50                                 Double.toString( result ) );
51                 }
52                 catch ( DivideByZeroException exception ) {
53                     showStatus( exception.toString() );
54                 }
55             }
56
57             return true;
58         }
59
60         // Definition of method quotient. Used to demonstrate
61         // throwing an exception when a divide-by-zero error
62         // is encountered.
63         public double quotient( int numerator, int denominator )
```

```
64     throws DivideByZeroException
65     {
66         if ( denominator == 0 )
67             throw new DivideByZeroException();
68
69         return ( double ) numerator / denominator;
70     }
71 }
```

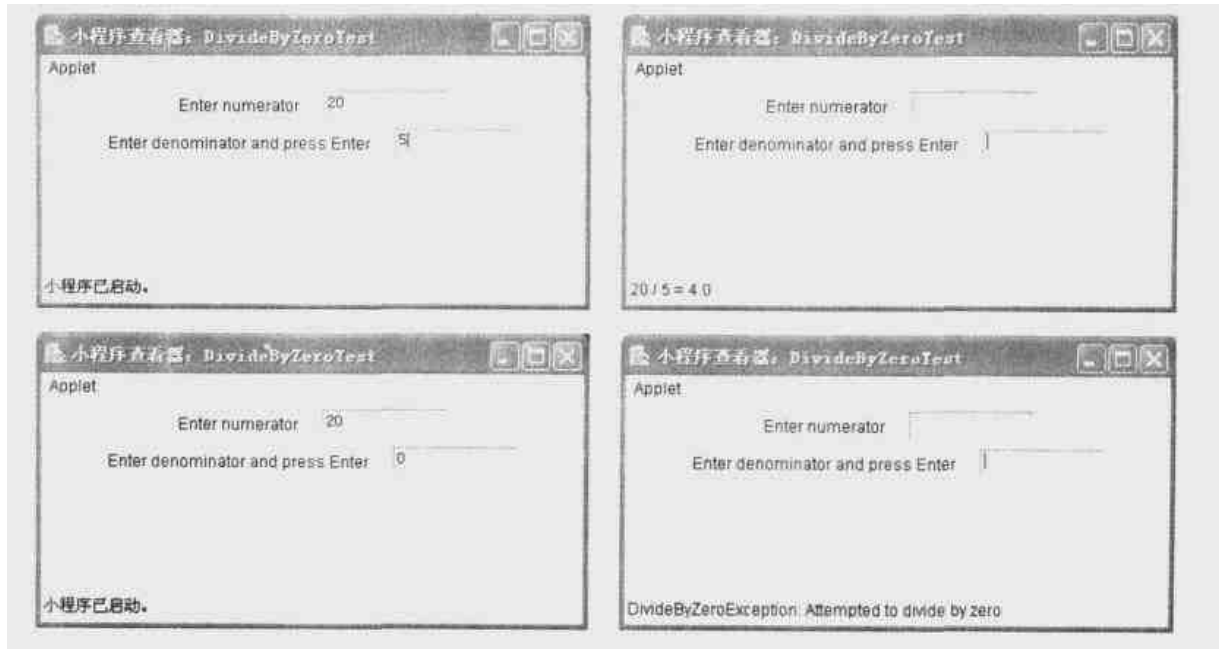


图 12.1 一个异常处理的简单实例——除数为零

考察四个输出窗口中的两个运行实例。前两个窗口中显示的是一次成功的执行。在后两个窗口中，输入的分母值为 0，程序检测出这个错误，因此抛出了一个异常，并显示相应的诊断信息。

首先，我们必须找到一个适于处理除零操作的 `Exception` 类，否则就必须创建一个异常类。在查找 `java.lang` 软件包的各个 `Exception` 类时，我们发现了一个完整的 `RuntimeException` 类集合，其中最适合我们需要的是 `ArithmeticException` 类。这个类可以直接使用，不过在本例中我们没有直接使用，而是创建了一个更特殊的 `DivideByZeroException` 类（第 1 行～第 11 行）。

#### 编程技巧 12.6

将每种“严重运行故障”都和一个相应命名的 `Exception` 类联系起来，可以提高程序清晰度。

`DivideByZeroException` 类是 `ArithmeticException` 类的扩展类。在它的构造函数（第 7 行～第 10 行）中，使用 `super` 方法来调用 `ArithmeticException` 的构造函数，传入的参数是字符串“Attempted to divide by zero”。

现在考虑 `DivideByZeroTestApplet`（第 12 行～第 71 行）。在 `init` 方法中（第 24 行～第 36 行），建立了一个带有两个 `Label` 和两个 `TextField` 的图形用户界面。

用户输入分母值后，就在 `action` 方法（第 38 行～第 58 行）中判断产生事件的是否为第二个文本字段（第 41 行），如果是，则表示所有的输入完成。两个文本字段中的内容将分别读入变量 `number1` 和 `number2` 中。

接着，在 `action` 方法中加入一个 `try` 程序块，将可能抛出异常的代码放到该块中。在 `try` 程序块中，可能产生错误的除法并没有显式地出现；但是调用了 `quotient` 方法，该方法将完成除法功能。

在 `quotient` 方法（第 60 行 ~ 第 70 行）中抛出了 `DivideByZeroException` 对象。一般情况下，通过执行 `try` 程序块中显式给出的代码，或是调用一个方法，甚至进行深层嵌套的方法调用，都可以使错误展现出来。

紧跟在 `try` 程序块后面的是一个 `catch` 程序块（第 52 行 ~ 第 54 行），其中包含 `DivideByZeroException` 的异常处理程序。一般情况下，当 `try` 程序块中抛出一个异常时，该异常将由与其类型相匹配的某个 `catch` 程序块所捕获。在图 12.1 中，`catch` 程序块指定了它将捕获的异常对象的类型为 `DivideByZeroException`；该类型和 `quotient` 方法中所抛出对象的类型是相匹配的。本例的异常处理程序只是简单地显示了一条错误信息，但事实上异常处理程序比这要复杂得多。执行完异常处理后，程序控制就转移到最后一个 `catch` 程序块之后的第一条语句上（本例中只包括一个 `catch` 程序块，但实际上可以含有多个 `catch` 块）。

接着，执行 `catch` 程序块之后的第一条语句。在图 12.1 中，执行了一条 `return` 语句，并返回 `true` 值。当然，如果在该程序运行时 `try` 程序块中的代码没有抛出异常，那么就跳过 `catch` 程序块，程序也可以继续向下执行。

#### 测试与调试提示 12.3

一个程序若采用了异常处理，就可以在解决问题之后继续执行。这有助于提高应用程序的鲁棒性，特别是对于那些任务关键的计算或关键的商业应用的计算。

现在我们分析一下 `quotient` 方法。当 `if` 语句判断 `denominator` 为零时，就调用 `throw` 语句，创建并抛出一个新的 `DivideByZeroException` 对象。该对象将由 `try` 程序块后面的 `catch` 程序块（指定匹配类型为 `DivideByZeroException`）所捕获，它由 `catch` 程序块的 `exception` 参数（第 52 行）接收，并通过 `toString` 方法将 `exception` 转换成一个字符串，然后使用 `showStatus` 方法将该字符串显示出来。

如果 `denominator` 不为零，那么不会抛出任何异常，程序将执行除法操作，然后将结果返回到 `try` 程序块中 `quotient` 方法的调用处（第 48 行）。接着使用 `showStatus` 方法显示算式和结果。程序将跳过 `catch` 程序块，`action` 方法将执行“`return true;`”语句。

注意，当 `quotient` 方法抛出了 `DivideByZeroException` 异常时，`quotient` 方法将终止执行。这样会对所有对象设置标记，用于无用单元回收处理；并且在进行无用单元回收处理之前，还将运行这些对象的终止函数。如果抛出了这种异常，那么 `try` 程序块在执行第 49 行的 `showStatus` 语句之前也会终止。另外，如果在 `try` 程序块中抛出异常之前创建了一些自动的对象，那么也将对这些对象设置标记，用于无用单元回收处理；并且在进行无用单元回收处理之前，还将运行它们的终止函数。

## 12.6 try 程序块

在一个 `try` 程序块中抛出的异常，通常是由紧跟在该 `try` 程序块后面的 `catch` 程序块中的异常处理程序捕获的。

```
try {  
    ...  
}  
catch ( ){  
    ...  
}
```

一个 `try` 程序块后面可以跟 0 个或多个 `catch` 程序块。

### 常见编程错误 12.2

在 try 程序块和相应的 catch 处理程序之间加入其他代码会产生语法错误。

如果执行一个 try 程序块时没有抛出任何异常,那么就跳过所有的异常处理程序,程序控制将转移到最后一个异常处理程序之后的第一条语句,如果在最后一个 catch 程序块后面包含一个 finally 程序块,那么不管是否抛出了异常都将执行该 finally 块中的代码。

### 测试与调试提示 12.4

在 finally 程序块中完成资源释放 (try 程序块所需的) 是非常合适的。这是避免资源泄漏的一种有效方法。

## 12.7 抛出异常

执行 throw 语句意味着发生了异常,这通常称为抛出异常。throw 语句需要指定要抛出的对象。throw 的操作数可以是 Throwable 类的任何派生类。Throwable 类的两个直接子类是 Exception 和 Error。Error 是指那些特别严重的系统问题,一般不应捕获,Exception 是由应捕获的问题引起的,可以在程序执行过程中进行处理,从而使程序更加强大。如果 throw 的操作数是 Exception 类的一个对象,那么我们将它称为异常对象。

### 测试与调试提示 12.5

当在任何一個 Throwable 对象上调用 toString 方法时,它的返回值是一个字符串,以提供给构造函数;如果未提供 String 则返回类名。

### 测试与调试提示 12.6

如果必须要传递有关导致异常的一些信息,则可以将该信息放到抛出的对象中。然后,该信息就会通过 catch 程序块的参数名而被引用。

### 测试与调试提示 12.7

一个对象可以不包含任何传递信息而将其抛出;此时,与其同类型的已抛出异常将为异常处理程序提供足够的信息。

当抛出一个异常时,控制程序将退出当前的 try 程序块,进入该 try 程序块后面的某个相应的 catch 处理程序中 (如果存在)。抛出点很可能位于 try 程序块的某个深层嵌套的作用域中;这时,程序控制仍然是进入 catch 处理程序。也可能抛出点是位于某个深层嵌套的方法调用,此时程序控制依然是进入 catch 处理程序。

try 程序块本身可以不包含错误检测语句和 throw 语句,但它所引用的内容将会执行构造函数中的错误检测代码,并且可能抛出异常。比如,在 try 程序块中为一个数组对象排列下标,如果排列了一个无效的数组下标,那么系统就会抛出 ArrayIndexOutOfBoundsException 异常。任何方法调用都可以引用可能抛出异常的代码,或调用其他能够抛出异常的方法。

尽管异常可以终止整个程序的执行,但它不必这么做。不过,一个异常至少会将产生该异常的那个程序块终止。

## 12.8 捕获异常

异常处理程序包含在 catch 程序块中。每个 catch 程序块都以关键字 catch 开头,后跟一对圆括

号, 括号中为类名(指定要抛出的异常的类型); 再后面是参数名, 可以通过该参数引用处理程序捕获的对象; 然后是一段程序代码, 用于描述异常处理功能。捕获到一个异常后, 就执行 catch 程序块中的代码。

#### 常见编程错误 12.3

如果假设异常处理之后, 程序控制将返回到 throw 后的第一条语句, 那么将导致逻辑错误

#### 常见编程错误 12.4

将 catch 程序块的参数设置成多个(由逗号分开)则会产生语法错误。一个 catch 程序块只能有一个参数。

#### 常见编程错误 12.5

如果两个不同的 catch 程序块(均和某个 try 程序块相关)都用于捕获同一类型的异常, 那么将会产生语法错误。

下面的 catch 语句表示将捕获所有的异常:

```
catch( Exception e )
```

#### 常见编程错误 12.6

将 catch( Exception e )语句放在其他 catch 程序块前面, 则会阻止这些程序块的执行; 该语句应放在各个异常处理程序的最后, 否则会产生语法错误。

#### 软件工程视点 12.6

如果使用 catch( Exception e )语句来捕获所有的异常, 那么可以利用 instanceof 运算符来判断异常的类型。例如, 如果对象 x 是 Y 类的一个实例, 那么 boolean 条件 “x instanceof Y” 将返回 true 值, 否则返回 false 值。

对于某个特殊的捕获对象, 很可能没有对应的处理程序, 这就需要在下一层 try 程序块中继续查找相应的处理程序。即使不断地进行查找, 最后也很可能发现程序中根本没有处理程序与该类型的捕获对象相匹配。这时, 没有基于 GUI 的应用程序将会终止, 而 applet 和基于 GUI 的应用程序则将返回正常的事件处理。

对于某一类型的异常, 很可能有几个异常处理程序都与之相匹配, 发生这种情况通常有几个原因。首先, 异常处理可能是用于“全部捕获”的 catch( Exception e ), 它将捕获任何异常。其次, 由于继承关系, 子类对象既可以由它本身的处理程序捕获, 也可以由它某个父类的处理程序捕获。如果发生这种情况, 那么在执行时将由所遇到的第一个异常处理程序与之进行匹配。

#### 软件工程视点 12.7

如果有几个处理程序匹配同一类型的异常, 并且每个处理程序的处理功能都不相同, 那么这些处理程序的顺序将影响该异常的处理方式。

#### 常见编程错误 12.7

如果捕获超类对象的 catch 语句放在捕获该超类的某个子类对象的 catch 语句之前, 则会产生语法错误。

有时, 在一个程序中可以同时处理很多关系密切的异常。程序员可以为一组异常只提供一个异常类及一个 catch 处理程序, 而不必为其中的每个异常都提供一个单独的类及相应的 catch 处理程序。当某个异常发生时, 可以根据不同的实例数据(比如一个类型码)创建该异常对象。catch 处理程序将检查该数据, 以判断异常的类型。实际上, 在 Java 这样的面向对象的编程语言中, 并不提倡使用这种编程风格, 因为利用继承可以将这种情况处理得更好。

在默认情况下, 如果某个异常没有找到相应的处理程序, 那么对于一个没有基于 GUI 的应用程

序来说,该程序将会终止执行。尽管这在原则上是正确的,但在使用其他不带异常处理功能的程序语言时,程序员却不习惯于这种处理方式。程序员会任由错误发生,然后再继续执行程序,这很可能使“不良影响”继续下去。

一个 try 程序块后面可以跟有几个异常处理程序段,这一点很像 switch 语句,但是并没有使用类似“break”的语句来退出异常处理程序。每个 catch 程序块都定义了一个特殊的作用域,这如同在 switch 语句中,针对每一种情况都有相应的作用域。

在异常处理程序中,不能访问在其相应的 try 程序块中定义的对象,因为当异常处理程序开始执行时,已经终止了 try 程序块。

如果在执行某个异常处理程序时又抛出了一个异常,那么会发生什么情况呢?因为当异常处理程序开始运行时,出现异常的 try 程序块已终止了,所以此时出现的异常就必须由原 try 程序块的外层程序进行处理。外层的 try 程序块将监视并处理在内层 try 程序块的 catch 处理程序中产生的错误。

异常处理程序有很多种不同的形式。它们可以简单地重抛出异常(下一节将介绍如何重抛出异常);可以通过抛出另一种不同类型的异常来转换异常的类型;可以在最后一个异常处理程序之后,完成任何必要的恢复工作并使程序继续执行;可以根据错误产生时的情况来判断错误的原因,然后进行解决,并重新调用原来曾产生该异常的方法;并且可以简单地向运行环境返回一些状态值;等等。

试图在 catch 处理程序中通过执行 return 语句来返回抛出点是不可能的。这样的 return 操作只能返回到包含该 catch 程序块的方法的调用方法中。而且,由于抛出点所在的程序块已经终止,因此想要通过 return 语句返回该抛出点是没有任何意义的。

#### 软件工程视点 12.8

传统的控制流不使用异常的另一个原因,是这些“额外的”异常可能会“掩盖”那些真正的错误类型的异常。程序员很难跟踪这么大量的异常情况,异常情况应该是不常见的。

#### 常见编程错误 12.8

假定在一个 catch 处理程序中抛出的异常,将由该处理程序或与同一个 try 程序块相关的其他处理程序来处理,则会产生逻辑错误。

## 12.9 重抛出异常

在用于捕获异常的 catch 处理程序中,很可能处理程序本身不能处理该异常,或是需要其他的 catch 处理程序来处理该异常。这时,可以在接收 Exception e 的处理程序中,通过下面的语句来重抛出该异常:

```
throw e;
```

该 throw 语句将把异常 e 重抛出到下一层的 try 程序块中。

即使在一个处理程序中能够处理某个异常,而且不管是否进行了处理,该处理程序仍然可以向外部重抛出该异常,从而进行进一步处理。一个重抛出的异常由下一层的 try 程序块检测,并由该层的 try 程序块的某个异常处理程序进行处理。

## 12.10 throws 子句

在一个方法中,可以利用 throws 子句列出该方法所能抛出的异常:



```
int g( float h )throws a,b,c
{
    //method body
}
```

在方法的声明部分,可以利用throws子句来指定该方法所抛出的异常的类型。一个方法可以抛出指定类的对象,或是子类的对象

有时,发生异常是由于某些本来可以避免的事情出现错误,这种情况称为运行时异常,它们是由 RuntimeException 类派生的。例如,如果程序中试图访问一个越界的数组下标,则会抛出一个 ArrayIndexOutOfBoundsException (从 RuntimeException 类派生而来)类型的异常。在程序中显然可以避免这个问题;因此,这是一个运行时异常。

当程序中已声明了一个对象引用而没有实际地创建该对象时,则会产生一个运行时异常。这种使用null引用的操作会抛出一个 NullPointerException 异常。很明显,在程序中完全可以避免这种情况;因此,它也是一个运行时异常。

另一种运行时异常是无效的强制类型转换操作,它会抛出一个 ClassCastException (参见第10章)异常。

还有很多种不是 RuntimeException 的异常,其中两种最常见的是 InterruptedException (参见第13章)和 IOException (参见第15章)。

并非所有能抛出的错误和异常都需要使用 throws 子句列出来。Error 就不必列出来, RuntimeException (可避免的异常)也不必列出来。Error 是指严重的系统问题,几乎在任何地方都会发生,大多数用户没有必要处理这些问题。RuntimeException 异常应直接进行处理,而不要把它们传送到程序的其他部分再进行处理。由一个方法显式抛出的非 RuntimeException 异常,以及由该方法所调用的方法抛出的非 RuntimeException 异常都必须列在该方法的 throws 子句中。

#### 软件工程视点 12.9

如果一个方法将抛出一个非 RuntimeException 异常,或将调用一个抛出非 RuntimeException 异常的方法,那么必须在该方法的 throws 子句中声明这些异常,或在该方法的某个 try/catch 程序块中进行处理。

Java 将异常区分为“经过检查的”Exception、“未经检查的”Exception 以及 Error。一个方法的“经过检查的”异常必须列在该方法的 throws 子句中。因为几乎任何方法都会抛出 Error 和 RuntimeException 异常,如果要求把这些异常也列到 throws 子句中,则会给程序员带来很大麻烦;这些异常不需要列在 throws 子句中,因此将它们称为“未经检查的”。一个方法所能抛出的所有非 RuntimeException 异常都必须列在该方法的 throws 子句中,因此将其称为“经过检查的”异常。

#### 常见编程错误 12.9

如果在一个方法中抛出了一个“经过检查的”异常,但该异常未列在该方法的 throws 子句中,则会产生语法错误。

#### 常见编程错误 12.10

试图在一个没有 throws 子句的方法中抛出一个“经过检查的”异常,则会产生语法错误。

#### 软件工程视点 12.10

如果要在一个方法中调用其他将显式抛出“经过检查的”异常的方法,那么必须将这些异常列在该方法的 throws 子句中,除非在该方法中直接捕获这些异常。这就是 Java 的“捕获或声明”需求。

#### 常见编程错误 12.11

如果一个子类方法重写了一个超类方法,那么当列在子类方法的 throws 子句中的异常数目多于超类方法

时，则会产生错误。子类方法的 throws 清单应是超类方法的 throws 清单的子集。

Java 的“捕获或声明”需求要求程序员必须捕获每一个“经过检查的”异常，或者将这些异常放到某个方法的 throws 子句中。当然，如果要将一个“经过检查的”异常放到 throws 子句中，那么必须使用其他方法对该异常进行处理。如果程序员觉得某个特殊的“经过检查的”异常是不可能发生的，那么可以捕获该异常，并且不做任何处理，这样可以避免必须在后面处理它的麻烦。但是，当某个程序发生了异常时，正确地处理该异常就变得很重要。

#### 测试与调试提示 12.8

不要认为简单地捕获异常而不进行任何处理，就可以满足 Java 的“捕获或声明”需求。异常通常都是非常严重的，因此必须进行处理，不能忽略。

#### 测试与调试提示 12.9

通过使用 throws 子句，Java 编译器将强制程序员处理这些可能抛出的异常，这有助于避免程序中出现的一些小问题。程序员往往容易忽视这些小问题，因此不对它们进行处理。

#### 软件工程视点 12.11

如果子类方法没有重写相应的超类方法，那么这些方法的异常处理操作与其超类方法是一致的。如果一个子类方法重写了超类方法，那么该子类方法中 throws 子句所包含的异常个数则不能多于其超类方法中 throws 子句所包含的异常个数。

#### 常见编程错误 12.12

Java 的编译器要求一个方法要么捕获它所抛出的所有“经过检查的”异常（可以直接在该方法中捕获，也可以间接地调用其他方法来捕获），要么将这些“经过检查的”异常引发到其他方法中；否则，Java 编译器就会产生语法错误。

#### 测试与调试提示 12.10

假设一个方法要抛出某个超类的所有子类对象，有些程序员可能认为在 throws 子句中只需列出超类即可；实际上，所有的子类都应该显式地列出来。这样，就可以着重关注那些必须进行处理的异常，这有助于避免由于使用通用的方法来处理异常所导致的错误。

图 12.2 到图 12.7 按继承层次列出了 Java 的 Error 和 Exception，其中图 12.2 中给出了 Java 的 Error 继承层次。大多数 Java 程序员都不必关注 Error，它们虽然是严重的错误，但却很少发生。

---

#### The java.lang package errors

---

Error (all in java.lang except for AWTError, which is in java.awt)

---

```

LinkageError
  ClassCircularityError
  ClassFormatError
  IncompatibleClassChangeError
    AbstractMethodError
    IllegalAccessError
    InstantiationException
    NoSuchFieldError
    NoSuchMethodError
  NoClassDefFoundError
  UnsatisfiedLinkError
  VerifyError
  ThreadDeath

```

```

VirtualMachineError (Abstract class)
    InternalError
    OutOfMemoryError
    StackOverflowError
    UnknownError
AWTError (in java.awt)

```

图 12.2 java.lang 软件包中的 Error

图 12.3 的内容非常重要，因为它列出了几乎所有的 `JavaRuntimeException`。尽管 Java 程序员没有把这些异常列在 `throws` 子句中，但是在 Java 应用程序中仍然经常捕获和处理这些异常。除了两个 `RuntimeException` 异常之外，其他所有的 `Java RuntimeException` 异常都在 `java.lang` 软件包中。

```

The java.lang package exceptions
Exception
    ClassNotFoundException
    ClassNotSupportedException
    IllegalAccessException
    InstantiationException
    InterruptedException
    NoSuchMethodException
    RuntimeException
        ArithmeticException
        ArrayStoreException
        ClassCastException
        IllegalArgumentException
            IllegalThreadStateException
            NumberFormatException
        IllegalMonitorStateException
        IndexOutOfBoundsException
            ArrayIndexOutOfBoundsException
            StringIndexOutOfBoundsException
        NegativeArraySizeException
        NullPointerException
        SecurityException

```

图 12.3 java.lang 软件包中的异常

图 12.4 中列出了 Java 的另外两个 `RuntimeException` 数据类型，我们在第 18 章讨论 `Vector` 类时将遇到这些异常。`Vector` 是一个动态数组，它可以增大或缩小，从而适应程序的可变存储需求。

```

The java.util package exceptions
Exception
    RuntimeException
        EmptyStackException
        NoSuchElementException

```

图 12.4 java.util 软件包中的异常

图 12.5 中列出了 Java 的 `IOException` 异常，它们都是在输入/输出和文件处理过程中可能产生

的“经过检查的”异常。

```
The java.io package exceptions
Exception
    IOException
        EOFException
        FileNotFoundException
        InterruptedIOException
        UTFDataFormatException
```

图 12.5 java.io 软件包中的异常

图 12.6 中列出了 java.awt 软件包中唯一的一个异常 AWTException，这是一个“经过检查的”异常，可以通过各种抽象窗口化工具箱（AWT）的方法将其抛出。

```
The java.awt package exceptions
Exception
    AWTException
```

图 12.6 java.awt 软件包中的异常

图 12.7 中列出了 java.net 软件包中的 IOException 异常，它们都是“经过检查的”异常，表示各种网络问题。

```
The java.net package exceptions
Exception
    IOException
        MalformedURLException
        ProtocolException
        SocketException
        UnknownHostException
        UnknownServiceException
```

图 12.7 java.net 软件包中的异常

## 12.11 构造函数、终止函数和异常处理

首先让我们讨论一个前面已经提到，但是还没有得到满意解决的问题：如果在构造函数中检测出了错误，将会发生什么情况？这个问题的关键是构造函数不能返回值，那么怎样才能让程序知道已无法正确构造该对象呢？一种解决方案是直接返回这个不正确的对象，然后在后面要使用该对象的部分中进行适当的判断，以确定这个对象实际上是错的。另一种解决方案是在构造函数外设置一些全局变量，但这并不是一个良好的编程习惯。比较好的处理方法是抛出一个异常，将构造函数的错误信息传递到外部，并处理该错误。

在构造函数中抛出的异常，使得在当前构造函数中建立的对象都将被设置标记，用于最终的无用单元回收处理，每一个对象在回收处理之前都将调用它的终止函数。另外，由于 Java 无法保证回收处理对象的顺序，因此也就无法保证调用终止函数的顺序。

## 12.12 异常和继承

各种异常类都可以从一个通用的超类派生。如果catch程序块可以捕获一个超类类型的异常对象，那么它也可以捕获该超类所有子类的异常对象。这样，就可以允许相关错误的多态处理了。

在异常上使用继承，可以使异常处理程序能够以一种简洁的表示方法来捕获各种相关的错误。可以在程序中分别捕获每一个子类异常对象，但更简洁的方法是直接捕获超类的异常对象。当然，仅当所有子类对象的处理方法都一致时才有意义。否则，就得捕获子类的异常。

### 测试与调试提示 12.11

如果程序员忘记对一个或多个子类类型进行显式测试，那么分别捕获子类异常对象时就会导致错误；而捕获超类对象则可确保所有子类对象都能被捕获。

## 12.13 finally 程序块

如果程序中使用了某些资源，那么最后必须将这些资源显式地释放回系统，以避免所谓的“资源泄漏”。在C和C++这样的编程语言中，最常见的一种资源泄漏是内存泄漏。在Java中，由于它实现了自动的无用内存单元回收功能，因此避免了大多数的内存泄漏。但Java还包括很多其他种类的资源泄漏。

### 软件工程视点 12.12

finally程序块中通常都包含一些用于资源释放（相应的try程序块所需要的）的代码，这是避免资源泄漏的一种有效方法。例如，在finally程序块中应该将try程序块（抛出了异常）中所有打开的文件都关闭。

### 测试与调试提示 12.12

实际上，Java并没有彻底地消除资源泄漏。只有对某个对象不再存有引用时，Java才会对该对象进行无用单元回收处理。因此，当程序员错误地对某个对象保持引用时，就会发生内存泄漏。大多数的内存泄漏问题都是由Java的无用单元回收处理功能解决的。

finally程序块是可选的，如果存在finally程序块，那么就将其放在try程序块的最后一个catch程序块的后面，如下所示：

```
try {
    statements;
    resource-acquire statements;
}
catch( AKindOfException ex1 ) {
    exception-handling statements;
}
catch ( AnotherKindOfException ex2 ){
    exception-handling statements;
}
finally {
    statement;
    resource-release statement ;
}
```

无论try程序块或与它相应的catch程序块中是否抛出了异常，Java都会确保执行finally程序块

(如果存在)。即使通过 `return`、`break` 或 `continue` 语句退出了 `try` 程序块,那么也会执行 `finally` 程序块(如果存在)。释放资源的代码放在 `finally` 块中。假设在 `try` 程序块中分配了一个资源,如果没有发生异常,则将跳过 `catch` 处理程序,直接转移到 `finally` 程序块中执行;并且释放资源,然后再继续执行 `finally` 程序块后面的第一条语句。

如果发生了异常,那么就跳过 `try` 程序块的其余部分,并在某一个 `catch` 处理程序中处理该异常(如果存在),然后仍然转移到 `finally` 程序块中执行;并且释放资源,最后继续执行 `finally` 程序块后面的第一条语句。

如果在 `try` 程序块中抛出的异常无法由某个 `catch` 处理程序捕获,那么就跳过 `try` 程序块的其余部分(因为终止了 `try` 程序块),并转移到 `finally` 程序块将资源释放。然后将该异常沿着调用链向外传递,直到某个方法捕获并处理该异常。如果没有任何方法能够处理该异常,那么一个没有基于 GUI 的应用程序就会终止。

如果在一个 `catch` 处理程序中抛出了一个异常,那么 `finally` 程序块仍将执行,而且该异常会沿着调用链向外传递,以便某个方法能够捕获并处理该异常。

图 12.8 的 Java 应用程序说明了即使在 `try` 程序块中没有抛出异常,也会执行 `finally` 程序块(如果存在)。该程序中有 3 个方法: `main`、`throwException` 和 `doesNotThrowException`。`main` 方法一开始执行就进入它的 `try` 程序块,并立即调用 `throwException` 方法。在 `throwException` 方法中,先抛出一个异常(第 23 行),然后捕获它(第 25 行),最后重抛出该异常。重抛出的异常将在 `main` 中处理,但首先要执行 `finally` 程序块(第 31 行~第 33 行)。该异常在 `main` 的 `try` 程序块(第 7 行~第 9 行)中已检测出来,并由 `catch` 程序块(第 10 行~第 13 行)进行处理。

```
1 // Fig. 12.8: UsingExceptions.java
2 // Demonstration of the try-catch-finally
3 // exception handling mechanism.
4 public class UsingExceptions {
5     public static void main( String args[ ] )
6     {
7         try {
8             throwException();
9         }
10        catch ( Exception e )
11        {
12            System.err.println( "Exception handled in main" );
13        }
14
15        doesNotThrowException();
16    }
17
18    public static void throwException() throws Exception
19    {
20        // Throw an exception and immediately catch it.
21        try {
22            System.out.println( "Method throwException" );
23            throw new Exception(); // generate exception
24        }
25        catch( Exception e )
26        {
27            System.err.println( "Exception handled in " +
28                               "method throwException" );
29            throw e; // rethrow exception for further processing
```

```

30         }
31         finally {
32             System.err.println( "Finally is always executed" );
33         }
34     }
35
36     public static void doesNotThrowException()
37     {
38         try {
39             System.out.println( "Method doesNotThrowException" );
40         }
41         catch( Exception e )
42         {
43             System.err.println( e.toString() );
44         }
45         finally {
46             System.err.println( "Finally is always executed." );
47         }
48     }
49 }

```

**输出结果:**

```

Method throwException
Exception handled in method throwException
Finally is always executed
Exception handled in main
Method doesNotThrowException
Finally is always executed.

```

图 12.8 try-catch-finally 异常处理机制

在 main 中，调用 doesNotThrowException 方法。由于 doesNotThrowException 方法的 try 程序块中没有抛出任何异常，因此跳过 catch 程序块，但仍然执行 finally 程序块。程序转移到 finally 程序块，因为 finally 程序块后面没有任何语句，所以控制将返回 main（终止）。

图 12.9 的 Java 应用程序说明，当在 try 程序块中抛出的异常没有被相应的 catch 程序块处理时，就将其传递到外层的 try 程序块中，并由相应的 catch 程序块（如果存在）处理。

可以看出，在很多情况下都将运行 finally 程序块，比如 try 程序块成功结束；抛出一个异常，然后又在本层的 catch 程序块中进行处理；抛出一个异常，但本层没有可用的 catch 程序块；或使用了一个程序控制语句，比如 return、break 或 continue。正常情况下，finally 程序块执行完后将根据进入该程序块的原因而产生相应的动作（我们将其称为 finally 的“后续动作”）。不过，finally 程序块也很可能执行一些不正常的“后续动作”。例如，如果在 finally 程序块中抛出了一个异常，那么后续动作将是在下一层的程序块中处理该异常。但是第一个异常将会丢失，这是很危险的。

```

1 // Fig. 12.9: UsingExceptions.java
2 // Demonstration of stack unwinding.
3 public class UsingExceptions {
4     public static void main( String args[] )
5     {
6         try {
7             throwException();
8         }

```

```
9         catch ( Exception e )
10        {
11            System.err.println( "Exception handled in main" );
12        }
13    }
14
15    public static void throwException() throws Exception
16    {
17        // Throw an exception and catch it in main.
18        try {
19            System.out.println( "Method throwException" );
20            throw new Exception(); // generate exception
21        }
22        catch( OtherException e )
23        {
24            System.err.println( "Exception handled in " +
25                               "method throwException" );
26        }
27        finally {
28            System.err.println( "Finally is always executed" );
29        }
30    }
31 }
32
33 class OtherException extends Exception {
34     public OtherException()
35     {
36         super( "Another exception type" );
37     }
38 }
```

**输出结果:**

```
Method throwException
Finally is always executed
Exception handled in main
```

图 12.9 异常的向外传递机制

**常见编程错误 12.13**

如果抛出一个异常后,没有可用的本层 catch 程序块,那么当程序执行到本层的 finally 程序块时,将会再抛出一个异常。如果发生了这种情况,那么第一个异常将会丢失。

**测试与调试提示 12.13**

要避免将可能抛出异常的代码放到 finally 程序块中。

**编程技巧 12.7**

Java 的异常处理机制主要是为了将错误处理代码与程序代码的主线分离开,以提高程序的清晰度。不要在每一个可能抛出异常的语句处都设置 try-catch-finally 功能,这会降低程序的可读性。确切地说,应该在程序中比较重要的部分设置 try 程序块,然后再分别设置几个 catch 程序块来处理不同的错误,最后添加一个 finally 程序块。

**软件工程视点 12.13**

作为一条原则,只要不再需要某项资源,就应该立即将其释放。这样就可以重新使用这些资源,从而提高程序的性能。



**软件工程视点 12.14**

如果一个 try 程序块含有相应的 finally 程序块, 那么即使通过 return、break 或 continue 语句退出了 try 程序块, 也将先执行 finally 程序块; 执行完后才会产生 return、break 或 continue 的结果。

## 12.14 使用 printStackTrace 和 getMessage 方法

异常是从 Throwable 类派生的。Throwable 类提供了一个 printStackTrace 方法, 用于打印方法调用堆栈的内容。在某个已被捕获的 Exception 对象中使用该方法, 就可以将方法调用堆栈的内容打印出来。这通常有助于测试和调试。在这一节中, 我们将通过一个例子来说明 printStackTrace 方法和另一个很有用的方法, getMessage。

**测试与调试提示 12.14**

所有的 Throwable 对象都包含一个 printStackTrace 方法, 用于打印该对象的调用堆栈内容。

**测试与调试提示 12.15**

如果一直未捕获某个异常, 那么就会运行 Java 的默认异常处理程序。该程序将显示异常的名称、创建该异常时提供的可选字符串, 以及一个完整的执行堆栈记录。该执行堆栈记录表示了完整的方法调用堆栈。这样, 程序员就可以看到该异常在文件 (即类) 和方法中的执行路径。

这个信息对于调试程序很有帮助。Exception 类有两个构造函数, 第一个构造函数没有参数, 如下所示:

```
public Exception ( )
```

第二个构造函数带有一个参数, 如下所示:

```
public Exception( String informationString )
```

参数 informationString 是对该类异常的描述信息, 可以利用 getMessage 方法获取保存在 Exception 中的 informationString。

**测试与调试提示 12.16**

Exception 类中有一个构造函数可以接收一个字符串, 使用这种形式的构造函数有助于确定异常的来源 (使用 getMessage 方法)。

图 12.10 说明了 getMessage 和 printStackTrace 方法的用法。getMessage 方法用于返回保存在某个异常中的描述字符串, printStackTrace 方法用于向标准错误流 (通常是命令行或控制台) 输出一个错误信息, 信息中包括该异常的名称、保存在该异常中的描述字符串以及异常抛出时还没有执行完的一系列方法 (即所有保留在当前的方法调用堆栈中的方法)。

在图 12.10 的程序中, main 方法调用 method1, method1 调用 method2, method2 又调用 method3。执行到这里时, 该程序的方法调用堆栈为:

```
method3  
method2  
method1  
main
```

最后调用的方法 (method3) 位于堆栈的顶部, 最先调用的方法 (main) 位于堆栈的底部。当 method3

抛出一个 `UserException` 异常时 (第 30 行), 调用堆栈就自动弹出栈中的内容, 直到出现捕获该异常的第一个方法为止 (即 `main` 方法, 因为它包含了一个 `catch` 处理程序, 用于处理 `UserException` 异常) 然后, 在 `catch` 处理程序中对 `UserException` 异常对象 `e` 使用 `getMessage` 方法和 `printStackTrace` 方法, 以产生输出信息。

```
1 // Fig. 12.10: UsingExceptions.java
2 // Demonstrating the getMessage and printStackTrace
3 // methods inherited into all exception classes.
4 public class UsingExceptions {
5     public static void main( String args[] )
6     {
7         try {
8             method1();
9         }
10        catch ( UserException e )
11        {
12            System.err.println( e.getMessage() +
13                               " \nThe stack trace is:" );
14            e.printStackTrace();
15        }
16    }
17
18    public static void method1() throws UserException
19    {
20        method2();
21    }
22
23    public static void method2() throws UserException
24    {
25        method3();
26    }
27
28    public static void method3() throws UserException
29    {
30        throw new UserException();
31    }
32 }
33
34 class UserException extends Exception {
35     public UserException()
36     {
37         super( "This is a user defined exception." );
38     }
39 }
```

**输出结果:**

```
This is a user defined exception.
The stack trace is:
UserException:This is a user defined exception.
    at UsingExceptions.method3(UsingExceptions.java:31)
    at UsingExceptions.method2(UsingExceptions.java:26)
    at UsingExceptions.method1(UsingExceptions.java:21)
    at UsingExceptions.main(UsingExceptions.java:8)
```

图 12.10 `getMessage` 和 `printStackTrace` 方法的使用

## 小结

- 常见的异常实例包括：内存溢出、数组下标越界、算法溢出、除数为零以及无效参数等。
- 异常处理是为处理同步错误而设计的；也就是说，异常随着程序的执行而产生。
- 通常，在故障诊断和故障处理不在同一位置的情况下使用异常处理。
- 异常不应该作为确定控制流的另一种机制。
- 异常处理应该用于处理来自某些软件组件的异常，这些软件组件包括方法、库、类等可能广泛应用的一些组件，在这些组件中由自己处理异常是没有意义的。
- 异常处理应该用在大型项目中，对于每个项目都以一种标准的方式进行错误处理。
- Java异常处理适用于在一个方法中能检测出错误但却不能处理错误的情况，在这样的方法中将抛出一个异常。如果一个异常与某个 catch 程序块中的参数类型相匹配，那么就执行该 catch 块中的代码。
- 程序员把可能产生异常的代码包含在一个 try 程序块中，try 程序块后面紧跟着一个或多个 catch 程序块。每个 catch 程序块都指定了它所能捕获和处理的异常类型。每个 catch 程序块都是一个异常处理程序。
- 如果一个 try 程序块中没有抛出任何异常，那么就跳过其中的异常处理程序，继续执行最后一个 catch 程序块之后的代码。如果有 finally 程序块，就执行 finally 程序块。
- 可以在方法的 try 程序块中抛出异常，也可以在 try 程序块中直接或间接调用的方法中抛出异常。
- 可以从异常抛出点向异常处理程序中传递信息，这些信息包括抛出对象本身的类型，或从发生异常的位置附近收集并放入抛出对象中的信息。
- throw 的操作数可以是 Throwable 类的任何派生类，Throwable 类的两个直接子类是 Exception 和 Error。
- RuntimeException 和 Error 异常通常称为“未经检查的”，非 RuntimeException 的异常通常称为“经过检查的”。必须将一个方法的“经过检查的”异常列在该方法的 throws 子句中。
- 异常由距它最近的（相对于抛出异常的 try 程序块）并且类型匹配的异常处理程序捕获。
- 如果抛出了一个异常，那么产生该异常的程序块将会终止。
- 在处理程序中，可以将一个异常对象重抛出到外层的 try 程序块中。
- catch( Exception e )用于捕获所有的 Exception。
- catch( Exception err )用于捕获所有的 Error。
- catch( Throwable t )用于捕获所有的 Exception 和 Error。
- 如果某个特殊的捕获对象没有与之相匹配的处理程序，那么就在下一层的 try 程序块中继续查找相应的处理程序。
- 对于某一类型的异常，可以根据类型依次查找与之相匹配的异常处理程序，第一个找到的异常处理程序将被执行。该处理程序执行完后，程序将继续执行最后一个 catch 程序块之后的第一条语句。
- 异常处理程序的排列顺序将影响异常的处理方式。
- 子类的异常对象既可以由它本身的处理程序捕获，也可以被它某个父类的处理程序捕获。
- 如果某个异常没有找到相应的处理程序，那么对于没有基于 GUI 的应用程序来说，将终止执行；而对于 applet 或基于 GUI 的应用程序来说，将返回正常的事件处理。
- 异常处理程序无法访问在它的 try 程序块中声明的变量，因为当异常处理程序开始执行时，

try 程序块已经终止了。处理程序所需的信息已经通过抛出对象向外传递。

- 异常处理程序可以重抛出一个异常, 或者通过抛出另一种不同类型的异常来转换异常的类型; 也可以在最后一个异常处理程序之后完成任何必要的恢复工作, 并使程序继续执行; 还可以根据错误发生时的情況来判断错误的原因, 然后消除该原因, 并重新调用原来曾产生异常的方法。另外, 还可以简单地向运行环境返回一些状态值。
- 捕获子类对象的处理程序应放在捕获超类对象的处理程序的前面。如果超类的处理程序放在前面, 那么该程序不但可以捕获超类对象, 还可以捕获该超类的各个子类对象。
- 当捕获到一个异常时, 在 try 程序块中分配的资源很可能没有释放。在 finally 程序块中可以释放这些资源。
- 有时, 可能用于捕获异常的 catch 处理程序本身不能处理该异常。这时, 只需简单地重抛出该异常即可。重抛出异常的语句格式是: 关键字 throw 后跟一个异常对象名。
- 即使在一个处理程序中能够处理某个异常, 仍然可以将该异常重抛出到外部, 以进行进一步的處理。一个重抛出的异常由下一层的 try 程序块检测, 并由位于该层 try 程序块后面的某个异常处理程序 (如果存在) 进行处理。
- 可以使用 throws 子句列出某个方法可能抛出的各个“经过检查的”异常。方法可以抛出指定的异常, 也可以抛出子类的异常对象。如果某个“经过检查的”异常没有列在 throws 子句中, 但却将其抛出, 则会产生语法错误。
- 使用继承关系, 可以使异常处理程序能够以一种简洁的表示方法捕获多种相关的错误。当然可以在程序中分别捕获每一个子类异常对象, 但更简洁的方法是直接捕获超类的异常对象。

## 术语

ArithmeticException

array exceptions 数组异常

ArrayIndexOutOfBoundsException

business-critical computing 关键的商业应用计算

catch a group of exceptions 捕获一组异常

catch all exceptions 捕获所有异常

catch an exception 捕获一个异常

catch block catch 程序块

catch( Exception e )

catch-or-declare requirement “捕获或声明”需求

checked Exceptions “经过检查的”异常

ClassCastException

declare exceptions that can be thrown 声明可以抛出的异常

default exception handler 默认异常处理程序

EmptyStackException

Error class Error 类

Error class hierarchy Error 类的层次结构

error handling 错误处理

exception 异常

Exception class Exception 类

Exception class hierarchy Exception 类的继承层次

exception handler 异常处理程序

exception handling 异常处理

exception object 异常对象

fault tolerance 容错

FileNotFoundException

finally block finally 程序块

getMessage() method of Throwable class Throwable 类的 getMessage() 方法

handle an exception 处理一个异常

IllegalAccess Exception

IncompatibleClassChangeException

instanceof operator instanceof 运算符

InstantiationException

InternalException

InterruptedException

|                                                  |                                                   |
|--------------------------------------------------|---------------------------------------------------|
| IOException                                      | RuntimeException                                  |
| library exception classes 库异常类                   | stack unwinding 堆栈解退                              |
| memory exhaustion 内存溢出                           | synchronous error 同步错误                            |
| mission-critical computing 任务关键的计算               | termination model of exception handling 异常处理的终止模式 |
| NegativeArraySizeException                       | throw an exception 抛出一个异常                         |
| NoClassDefFoundException                         | throw point 抛出点                                   |
| nonruntime exception 非运行时异常                      | throw statement throw 语句                          |
| null reference null 引用                           | Throwable class Throwable 类                       |
| NullPointerException                             | throws clause throws 子句                           |
| OutOfMemoryException                             | try block try 程序块                                 |
| resource leak 资源泄漏                               | unchecked Exceptions “未经检查的”异常                    |
| resumption model of exception handling 异常处理的恢复模式 | UnsatisfiedLinkException                          |
| rethrow an exception 重抛出一个异常                     |                                                   |

## 自测练习

- 12.1 列出5个常见的异常实例。
- 12.2 为什么异常处理技术不应该用于传统的程序控制？
- 12.3 为什么异常处理特别适用于处理那些由类库和方法所产生的错误？
- 12.4 什么是“资源泄漏”？
- 12.5 如果在try程序块中没有抛出异常，那么当该try程序块执行完后，程序将转移到什么位置继续执行？
- 12.6 如果发生了一个异常，但没有找到适当的异常处理程序，那么会发生什么情况？
- 12.7 给出使用catch(Exception e)的一个优点。
- 12.8 一个传统的applet或应用程序应该捕获Error对象吗？
- 12.9 如果同时有几个异常处理程序都匹配同一类型的抛出对象，那么会发生什么情况？
- 12.10 为什么程序员会将一个超类类型作为catch处理程序的类型，然后再抛出子类型的对象呢？
- 12.11 在一个catch处理程序中，如果不使用异常类的继承，那么应怎样处理具有相关类型的错误？
- 12.12 使用finally程序块的关键理由是什么？
- 12.13 抛出一个异常一定会导致程序终止吗？
- 12.14 在catch处理程序中抛出异常后会发生什么情况？
- 12.15 try程序块抛出异常后，在该程序块中已建立的自动对象会发生什么情况？

## 自测练习答案

- 12.1 内存溢出，数组下标越界，算法溢出，除数为零，无效参数。
- 12.2 a) 异常处理是为处理一些不常发生的情况而设计的（这些情况通常会导致程序终止），

- 因此编译程序并非一定要提供异常处理功能。
- b) 使用传统控制结构的控制流通常比使用异常更加清晰和有效。
  - c) 当发生了一个异常,并且在此之前所分配的资源还没有释放时,可能会因为调用堆栈没有弹出其中的内容而产生一些问题。
  - d) 可能会出现一些真正错误类型的“额外”异常,并且程序员将很难跟踪这么多的异常情况。
- 12.3 要想在类库和方法中实现满足所有用户惟一需求的错误处理是不可能的。
- 12.4 强制终止一个程序可能会导致永远占用而不释放某个资源,这样其他程序将无法申请该资源。
- 12.5 将跳过该 try 程序块的异常处理程序(在 catch 程序块中),并继续执行最后一个 catch 程序块后面的程序。如果存在 finally 程序块,就执行该程序块,然后再执行 finally 程序块后面的程序。
- 12.6 没有基于 GUI 的应用程序会终止; applet 或基于 GUI 的应用程序会恢复正常的事件处理。
- 12.7 catch (Exception e) 可以捕获在 try 程序块中抛出的任何类型的异常,使用该语句的好处是不会漏掉任何抛出的异常。
- 12.8 Error 通常都是与基本 Java 系统有关的一些严重问题。大多数程序都不处理 Error。
- 12.9 执行第一个与之相匹配的异常处理程序。
- 12.10 这是捕获相关类型异常的一个好方法,但必须小心使用。
- 12.11 为一组异常只提供一个 Exception 子类和一个 catch 处理程序。当发生异常时,可以根据不同的实例数据来创建该异常对象。catch 处理程序将检查该数据,以判断异常的类型。
- 12.12 使用 finally 程序块是防止资源泄漏的首选办法。
- 12.13 不一定,但抛出了异常的程序一定会终止。
- 12.14 该异常将由外层 try 程序块(如果存在)的某个 catch 处理程序(如果存在)来处理。
- 12.15 为这些对象设置标记,用于无用单元回收处理。在回收处理每个对象之前都要执行它的终止函数。

## 练习

- 12.16 在什么情况下应使用下面的语句:

```
catch(Exception e) {throw e;}
```

- 12.17 异常处理与传统错误处理方法相比,其优势是什么?
- 12.18 描述处理相关异常的一种面向对象的技术。
- 12.19 在本章之前我们已经发现,对构造函数检测出的错误进行处理有一些麻烦。试着解释一下为什么利用异常处理可以有效地处理构造函数中的错误。
- 12.20 假设一个程序抛出了一个异常,并开始执行相应的异常处理程序。现在,再假设在该异常处理程序中又抛出了一个同样的异常。这会导致无限循环吗?为什么?
- 12.21 通过继承关系可以创建一个异常超类和几个异常子类。编写一个程序,说明用于捕获超类异常的 catch 处理程序也可以捕获子类异常。
- 12.22 编写一个 Java 程序,说明当在 try 程序块中抛出了异常后,该程序块中创建的所有对象都要调用自己的终止函数。

- 12.23 编写一个Java程序,说明利用 `catch( Exception e )` 可以捕获各种不同的异常。
- 12.24 编写一个Java程序,说明异常处理程序的排列顺序是很重要的,将执行第一个与抛出的异常相匹配的处理程序。使用两种方式编译和运行该程序,以便通过不同的执行结果来表示执行了不同的处理程序。
- 12.25 编写一个Java程序,说明构造函数可以将有关该构造函数的错误信息传递到 `try` 程序块后面的异常处理程序中。
- 12.26 编写一个Java程序,说明重抛出异常的用法。
- 12.27 编写一个Java程序,说明带有 `try` 程序块的方法不必捕获该程序块中可能产生的每一个错误,有些异常可以由外层处理。

# 第13章 多线程

## 教学目标

- 理解多线程的概念
- 认识到多线程可以提高性能
- 理解创建、管理和终止线程的方法
- 理解一个线程的生命周期
- 学习几个线程同步的例子
- 理解线程优先级与调度
- 理解精灵线程和线程组

## 13.1 简介

通常，一个比较好的做事原则是“一次只做一件事情”并且“一次就做好”。但在现实世界中，事情并非这么简单。人体中有大量的活动都是并行或并发（本章将要介绍）完成的。比如，呼吸、血液循环和消化这三个活动可以并发地完成。所有的感觉——包括视觉、触觉、嗅觉、味觉和听觉——都可以并发地出现。在驾驶汽车时，可以并发地完成加速转弯、空气调节和播放音乐等功能。计算机的很多操作也是并发完成的。如今，在个人计算机上同时执行程序的编译、文件的打印以及从网络上接收电子邮件等是很常见的。

并发在我们的生活中是很重要的。尽管如此，大多数编程语言却不提供并发机制。更确切地说，这些编程语言一般只提供几种简单的控制结构，利用这些控制结构，一次只能执行一个动作。只有前一个动作完成后，才能开始执行下一个动作。目前，计算机上的这种并发通常都是利用操作系统的“原语操作”来实现的，而只有那些经验丰富的“系统程序员”才使用这些“原语操作”。

由美国国防部开发的Ada语言，为创建指挥和控制系统的厂商提供了广泛且有效的并发原语操作，但Ada语言在大学和商界没有得到广泛应用。

Java是唯一一种为PC机程序员提供广泛有效的、并发原语操作的通用编程语言。程序员可以在应用程序中加入线程，每个线程都可以完成程序的某一部分功能，并可以与其他线程并发执行。这种机制称为多线程，它使Java程序员拥有了C和C++程序员所不具备的能力，尽管Java曾是基于C和C++的。我们将C和C++称为单线程语言。

### 软件工程视点 13.1

很多语言都不提供内置的多线程机制（比如C和C++），因此必须调用操作系统的多线程原语操作。Java和这些语言不同，它本身就包含多线程原语操作（包含在java.lang软件包中的Thread类、ThreadGroup类和ThreadDeath类中）。这样，程序员在编写应用程序时就可以使用多线程。

下面，我们将讨论很多并发应用程序。在从WWW（万维网）上下载大型文件（比如音频或视频文件）时，我们并不愿意一直在等待，直到整个文件都下载完毕才将其读出来；因此我们可以启



动多个线程，一个线程用于下载，另一个用于播放或显示所下载的文件，这样就可以并发地执行这些操作或任务。为了避免播放时的不连贯，我们将协调各个线程，只有当内存中有了足够的信息能使播放线程运行起来时才启动播放线程。

另一个多线程的例子是Java的自动无用单元回收处理。在C和C++中，动态回收已分配内存的工作都由程序员来完成，而Java提供了一个无用单元回收处理线程，它能够自动地动态回收不再需要的内存空间。

#### 测试与调试提示 13.1

在C和C++中，程序员必须显式地提供动态回收内存的语句。如果没有回收内存（可能因为程序员忘了加上回收语句，也可能因为一个逻辑错误或异常而失去了程序的控制），则会导致“内存泄漏”，最终耗尽内存的可用空间，从而使程序终止。Java的自动无用单元回收处理功能可以消除绝大多数的内存泄漏情况，即导致悬挂对象（未被引用的对象）的情况。

Java的无用单元回收处理程序是一个低优先级的线程。当Java确定对某个对象不再有引用时，就对该对象设置无用单元回收处理标记。当处理器空闲，并且没有其他高优先级的线程运行时，就运行无用单元回收处理程序。不过，当系统发生内存溢出时，无用单元回收处理程序会立即启动。

#### 性能提示 13.1

Java的无用单元回收处理功能无法和最好的C/C++程序员所编写的动态内存管理程序相媲美，但对于程序员来说，它还是相当有效的，而且更安全。

#### 性能提示 13.2

将对象引用设置为null，即对该对象设置无用单元回收处理标记（如果再没有对该对象的其他引用），这将有助于保留系统的内存。在这样的系统中，一个自动对象不会超出其作用域，因为它所在的方法将长期执行。

编写多线程程序是需要一些技巧的。尽管人们通常可以并发地完成很多操作，但有时要想在并行的“思维链”之间进行跳转还是很困难。为了说明为什么多线程程序难于编写和理解，让我们来做一试验：打开3本书。首先从第一本书上读几个单词，然后再从第二本书上读几个单词，然后再从第三本书上读几个单词；接着返回第一本书，往下再读几个单词，反复下去。过不了多久，你很快就会感受到多线程执行的困难：一面要换书；一面要快速地阅读；一面要记住每本书的阅读位置；一面要把要读的书放到眼前，以便看清；一面要把不读的书推开；而在手忙脚乱地完成上述动作的同时，还必须要理解每本书的内容！

#### 性能提示 13.3

使用单线程应用程序的问题是在开始执行某个操作之前，可能必须要完成一长串其他操作。由于用户们觉得他们在使用Internet和WWW时花费了太多的等待时间，因此多线程立即受到欢迎。

尽管Java可能是世界上可移植性最好的编程语言，但它的某些部分仍然是依赖于运行平台的。尤其是早期实现的3个Java平台之间有很多差别，这3个平台分别是Solaris、Windows 95及Windows NT。

在早期的Solaris Java平台上，当运行某个具有给定优先级的线程时，如果没有更高优先级的线程处于就绪状态，就将该线程执行完。如果有更高优先级的线程就绪，就会发生“抢占”情况，也就是处理器将由高优先级的线程抢占过来，而原来正在运行的低优先级线程就必须进入等待状态。

在Windows 95和Windows NT上的32位Java版本中，线程是按照时间分片的方式执行的。也就是说，每个线程在处理器上都只能执行一段有限的时间（称为一个时间分片），当超过这段时间

时,该线程就会进入等待状态;同时,所有其他具有相同优先级的线程就会以轮转方式平等地获取自己的时间分片。当轮转一圈后,原来已进入等待状态的线程就会恢复执行。这样,在 Windows 95 和 Windows NT 上,一个正在运行的线程就可能遇到其他具有相同优先级的线程的抢占;而在早期的 Solaris 平台上,一个正在运行的线程只能由具有更高优先级的线程所抢占。在以后的 Solaris Java 系统中,将力求实现时间分片方式。

#### 可移植性提示 13.1

Java 的多线程是依赖于平台的。因此,一个带有多线程的 Java 应用程序在不同平台上的运行结果是不同的。

## 13.2 Thread 类: 线程方法介绍

在这一节中,我们将看到 Java API 中各种与线程相关的方法。本章的很多例子都使用了这些方法。如果读者想了解更多的细节,尤其是每种方法所引发的异常,可以直接查阅 Java API。

Thread 类有几个构造函数。下面的构造函数是创建一个名为 threadName 的线程:

```
public Thread( String threadName )
```

而下面的构造函数创建了一个名为 Thread*n* (即在 Thread 后跟一个数字,比如 Thread1, Thread2 等)的线程:

```
public Thread ( )
```

用于完成一个线程“实际功能”的代码放在 run 方法中。run 方法可以在 Thread 的一个子类中重写,也可以在一个 Runnable 对象中重写,Runnable 是一个重要的 Java 接口,我们将在 13.10 节中介绍。

在一个程序中,通过调用 start 方法(然后再调用 run 方法)可以启动一个新线程。当使用 start 方法启动了新线程后,程序控制立即返回调用者,然后新线程与调用者就开始并发地执行。如果使用 start 方法启动一个已经启动的线程,则会引发 IllegalThreadStateException 异常。

static sleep 方法带有一个参数,指定了当前执行线程的睡眠时间(单位是毫秒);当一个线程处于睡眠状态时,它不会争夺处理器,以便其他线程继续执行。这就为低优先级的线程提供了运行的机会。

interrupt 方法用于中断一个线程。如果某个线程已经中断,那么 interrupted 方法(static 方法)将返回 true 值,否则将返回 false 值。使用 isInterrupted 方法(一个非 static 方法)可以确定某个线程是否中断。

suspend 方法用于将执行的线程挂起,而 resume 方法用于恢复挂起的线程。如果使用 resume 方法恢复一个未挂起的线程,则该线程不会受到影响,可以继续执行。

stop 方法用于停止一个线程,并引发一个 ThreadDeath 对象。大多数用户都使用不带参数的 stop 方法。

如果某个线程已经调用了 start 方法,但是还没有调用 stop 方法,那么 isAlive 方法将返回 true 值。我们将在讨论了线程优先级和线程调度之后再详细讨论 yield 方法。

setName 方法用于设置线程的名称,getName 方法用于返回线程的名称。toString 方法则将返回一个字符串,其中包括线程的名称、线程的优先级以及线程所在的线程组。

static 的方法 currentThread 用于返回对当前 Thread 的一个引用。

join方法(带有一个参数,表示时间长度,单位为毫秒)用于等待目标线程接收终止信息,以执行当前线程。如果join方法没有参数或参数值为0毫秒,则表示当前线程将一直等待下去,直到目标线程结束才继续执行。这种等待是很危险的,它可能导致两个特别严重的问题:死锁和无限延迟。我们将在稍后讨论这两个问题。

#### 测试与调试提示 13.2

dumpStack方法用于调试多线程应用程序。程序中可以用static的dumpStack方法(通过Thread.dumpStack)打印当前线程的方法调用堆栈轨迹。

### 13.3 线程状态：一个线程的生命周期

一个线程在生存期间总是处于某一种状态(如图13.1所示)。假设一个线程在刚刚创建时称为“新生”状态,则该线程在调用start方法之前将一直处于“新生”状态;调用了start方法之后,该线程就进入“就绪”状态。当系统为线程分配了处理器时,具有最高优先级的就绪线程就将进入“运行”状态(即该线程开始运行)。当一个线程的run方法运行完毕,或调用了stop方法时,该线程就进入“停止”状态。一个停止的线程将被系统永久清除。

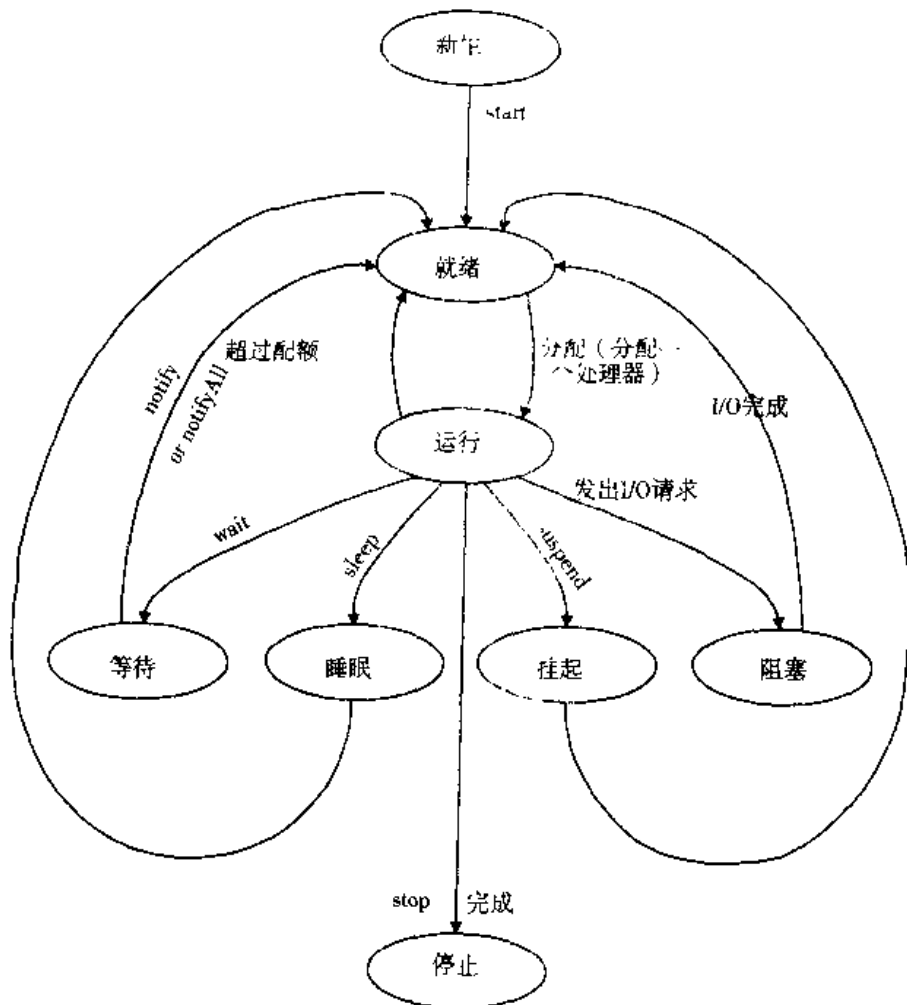


图 13.1 线程的生命周期

当一个正在运行的线程发出一个输入/输出请求时,就会进入“阻塞”状态。当输入/输出请求得到回应时,“阻塞”线程就会进入“就绪”状态。一个“阻塞”线程无法直接使用处理器(即使处理器空闲也无法使用)。

当调用一个“运行”线程的 `sleep` 方法时,该线程就进入“睡眠”状态。一个“睡眠”线程超过了睡眠时间之后就进入“就绪”状态。一个“睡眠”线程无法直接使用处理器(即使处理器空闲也无法使用)。

#### 性能提示 13.4

延迟线程的一种方法是执行一个循环若干次的空循环体,这称为“忙等待”,它会导致系统性能降低;因为处理器在忙着执行空循环体,其他的可执行线程就必须等待。一个更好的方法是调用该线程的 `sleep` 方法,“睡眠”线程不能直接使用处理器(即使处理器空闲也无法使用)。

当一个“运行”线程调用 `suspend` 方法时,该线程就进入“挂起”状态。当一个“挂起”线程的 `resume` 方法被另一个线程调用时,它就进入“就绪”状态。“挂起”线程不能直接使用处理器(即使处理器空闲也无法使用)。

当一个“运行”线程调用 `wait` 方法时,就进入“等待”状态,即进入一个与调用 `wait` 的对象相关的等待队列中进行等待。当另一个线程发出 `notify` 调用时,等待队列中的第一个线程将进入“就绪”状态。当另一个线程发出 `notifyAll` 调用时,等待队列中的每一个线程都将进入“就绪”状态。稍后在我们讨论监控器时,将更深入地介绍 `wait`、`notify` 和 `notifyAll` 方法。

当一个线程的 `run` 方法运行完毕或调用 `stop` 方法时,该线程会进入“停止”状态。`stop` 方法向该线程发出一个 `ThreadDeath` 对象, `ThreadDeath` 是 `Error` 类的一个子类。

#### 常见的编程错误 13.1

`ThreadDeath` 是一个 `Error`, 不是一个 `Exception`, 用户程序不应捕捉 `Error` 对象。如果非要捕捉一个 `ThreadDeath` 对象的话,一定要重引发它,否则该线程就无法“正确地”停止。

## 13.4 线程优先级与线程调度

每个 Java applet 或应用程序都是多线程的。每个 Java 线程都有一个优先级,范围在 `Thread.MIN_PRIORITY` (该常量的值为 1) 和 `Thread.MAX_PRIORITY` (该常量的值为 10) 之间。默认情况下,每个线程的优先级都为 `Thread.NORM_PRIORITY` (该常量的值为 5)。每个新线程都继承其父线程的优先级。

有些 Java 平台支持时间分片方式,有些则不支持。如果不采用时间分片方式,那么具有相同优先级的每个线程都会一直执行到结束,才会把处理器让给其他的同级线程执行。如果采用时间分片方式,那么每个线程都只能在一个短暂的处理器时间分片内执行。当时间分片结束时,即使线程还没有执行完,也要将处理器让给下一个具有相同优先级的线程。

Java 调度程序的功能是保证在任意时刻运行的线程都是最高优先级的,如果采用时间分片方式,那么调度程序就要确保具有相同优先级的每个线程,都能以轮转方式轮流运行时间分片(即这些线程都按时间分片方式运行)。图 13.2 说明了 Java 线程的多级优先级队列。在该图中,线程 A 和 B 以轮转方式轮流执行,每次执行一个时间分片,直到两个线程都执行完。接着执行线程 C,直到结束。然后,再以轮转方式轮流执行线程 D、E 和 F,直到这三个线程都执行完。注意,一个具有较高优先级的新线程可能会延迟低优先级线程的执行——很可能是无限的延迟。这种无限延迟通常称为互斥等待。

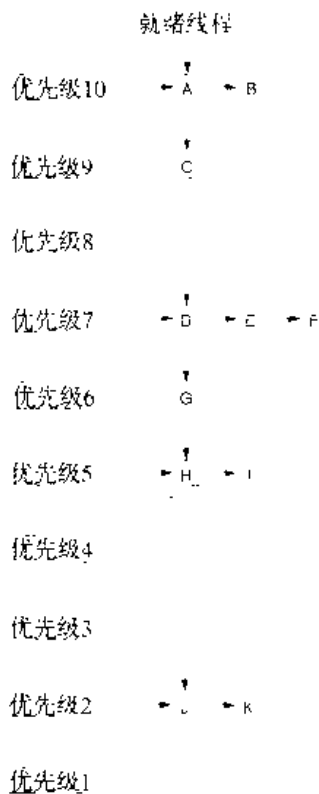


图 13.2 Java 线程的优先级调度

可以使用 `setPriority` 方法来调整一个线程的优先级, 该方法带有一个整型参数。如果参数值不在 1 到 10 的范围内, 那么 `setPriority` 方法将引发一个 `IllegalArgumentException` 异常。`getPriority` 方法用于返回线程的优先级。

一个线程可以调用 `yield` 方法来为其他线程提供执行的机会, 实际上无论何时, 只要某个具有较高优先级的线程进入就绪状态, 那么当前线程就会被抢占。因此, 一个线程不能对具有较高优先级的线程执行 `yield` 方法, 因为当具有较高优先级的线程进入就绪状态时, 当前线程已经被抢占了。类似地, `yield` 方法总是只允许优先级最高的就绪线程运行。所以, 如果在调用 `yield` 方法时只有较低优先级的线程处于就绪状态, 那么当前线程就变成优先级最高的线程, 它将继续执行下去。因此, 线程调用 `yield` 方法实际上是为与它优先级相同的线程提供运行机会。在采用时间分片方式的系统中, 这是没有必要的, 因为优先级相同的各线程将以轮转方式分别执行各自的时间分片 (或由于某些其他原因而失去处理器时间)。所以, `yield` 方法适用于采用非时间分片方式的系统, 在这样的系统中, 如果优先级相同的其他线程没有得到运行的机会, 那么当前线程会一直运行下去, 直到结束。

#### 性能提示 13.5

在采用非时间分片方式的系统中, 优先级相同的各个线程应定期地调用 `yield` 方法, 使它们的同级线程能够顺利地执行下去。

#### 可移植性提示 13.2

应保证所编写的 applet 在所有 Java 平台上都能运行, 从而提高 Java 的可移植性。

一个线程除非停止了, 或因为输入/输出而进入了阻塞状态, 或调用了 `sleep` 方法、`yield` 方法, 或被高优先级的线程抢占了, 或超出了时间分片, 否则该线程就会执行。如果恢复一个挂起线程, 或唤醒一个睡眠线程, 或等待 I/O 请求的线程得到回应, 或者在一个调用了 `wait` 的线程上调用了 `notify`

方法或 `notifyAll` 方法, 那么该线程就会成为就绪状态 (因此会抢占运行的线程)。图 13.3 的应用程序说明了一些基本的线程技术, 包括从 `Thread` 类创建一个派生类, 构造一个线程, 以及使用 `Thread` 类的 `sleep` 方法。程序中创建的每个线程都要睡眠一个随机的时间段 (长度在 0~5 秒之间), 然后再执行, 并显示它的名称。这个程序包含两个类——`PrintThread` 和 `PrintTest`。

```
1 // Fig. 13.3: PrintTest.java
2 // Show multiple threads printing at different intervals.
3
4 public class PrintTest {
5     public static void main( String args_ )
6     {
7         PrintThread thread1, thread2, thread3, thread4;
8
9         thread1 = new PrintThread( "1" );
10        thread2 = new PrintThread( "2" );
11        thread3 = new PrintThread( "3" );
12        thread4 = new PrintThread( "4" );
13
14        thread1.start();
15        thread2.start();
16        thread3.start();
17        thread4.start();
18    }
19 }
20
21 class PrintThread extends Thread {
22     int sleepTime;
23
24     // PrintThread constructor assigns name to thread
25     // by calling Thread constructor
26     public PrintThread( String id )
27     {
28         super( id );
29
30         // sleep between 0 and 5 seconds
31         sleepTime = (int) ( Math.random() * 5000 );
32
33         System.out.println( "Name: " + getName() +
34                             "; sleep: " + sleepTime );
35     }
36
37     // execute the thread
38     public void run()
39     {
40         // put thread to sleep for a random interval
41         try {
42             sleep( sleepTime );
43         }
44         catch ( InterruptedException exception ) {
45             System.err.println( "Exception: " +
46                                 exception.toString() );
47         }
48
49         // print thread name
50         System.out.println( "Thread " + getName() );
51     }
52 }
```

输出结果:

```
Name: 1; sleep: 429 /
Name: 2; sleep: 6 / 2
Name: 3; sleep: 2 /
Name: 4; sleep: 2090
Thread 3
Thread 2
Thread 4
Thread 1
```

图 13.3 睡眠随机时间段的多线程

从 Thread 类继承的 PrintThread 类包含一个实例变量 sleepTime、一个构造函数及一个 run 方法。实例变量 sleepTime 中保存的是一个随机的整数值,该值在创建 PrintThread 对象时进行设置。每个 PrintThread 对象都将睡眠一段时间(时间长短由 sleepTime 确定),然后再输出它的名称。PrintThread 构造函数带有一个 String 参数,代表线程的名称。通过该 String 调用超类的 Thread 构造函数,Thread 类的构造函数利用这个 String 来设置线程的名称。然后,PrintThread 构造函数设置 sleepTime 初值,该值为 0 到 5 000 (0~5 秒)的一个随机整数。接着,输出线程名称及 sleepTime 的值。当调用一个 PrintThread 的 start 方法(从 Thread 继承)时,该 PrintThread 对象就进入就绪状态。当系统为该 PrintThread 对象分配了一个处理器时,它就进入运行状态,开始执行它的 run 方法。在 run 方法中调用 sleep 方法(第 42 行),使该对象立即进入睡眠状态。当唤醒该对象时,它就又进入就绪状态,直到为其分配了一个处理器。当这个 PrintThread 对象再次进入运行状态时,它将输出自己的名称,然后终止 run 方法,最后该线程对象将进入停止状态。注意, sleep 方法可能会引发一个 InterruptedException 异常,所以必须在 try 程序块中调用 sleep 方法,或者在 run 方法的 throws 子句中声明该异常。

PrintTest 的 main 方法创建了 4 个 PrintThread 对象,然后每个对象都分别调用 Thread 类的 start 方法,使这 4 个对象都处于就绪状态。

## 13.5 线程同步

Java 利用监控器的手段(详细内容请参见练习 13.24 中引证的由 C.A.R.Hoare 于 1974 所写的论文)来完成同步操作。每个带有 synchronized 方法的对象都是一个监控器。监控器一次只允许一个线程对该对象执行 synchronized 方法。当调用某个对象的 synchronized 方法时,就锁定该对象,这也就是所谓的“获得锁”。如果在一个对象中存在几个 synchronized 方法,那么一次只能有一个 synchronized 方法是激活的;所有其他要调用 synchronized 方法的线程都必须等待。在一个 synchronized 方法执行结束后,就打开该对象上的锁,监控器会让调用 synchronized 方法的线程中优先级最高的那个线程继续执行。

一个执行 synchronized 方法的线程可以判断出自己无法再继续执行,因此自愿地调用 wait 方法进行等待,这样可以使该线程不再争用处理器和监控器对象。现在,该线程就在队列中等待,而其他的线程都在试图进入该对象。当一个执行 synchronized 方法的线程结束时,该对象会通知(notify 方法)某个等待的线程,并使其变为就绪状态。这样该线程就可以获得该监控器对象的开锁钥匙,并开始执行。这里的 notify 就像是发给等待线程的一个信号,告诉该线程它所等待的条件现在已满足了,因此有资格重新进入监控器。如果一个线程调用了 notifyAll 方法,那么所有正在等待该对象的线程就都有资格重新进入监控器(也就是将它们都设置为就绪状态)。请记住,在这些线程中,一

次只能有一个线程获得该对象的开锁钥匙。

#### 常见编程错误 13.2

最终必须使用 `notify` 方法显式地唤醒等待监控器对象的线程，否则该线程就会永远等待下去，这是死锁的一个简单例子。

#### 测试与调试提示 13.3

确保每调用一次 `wait` 方法，就要相应地调用一次 `notify` 方法，以便最终结束等待

#### 性能提示 13.6

在获得多线程程序正确性的前提下使用同步操作会降低程序的运行性能，因为线程会进行等待。不过，如果程序不正确，那么性能再好也是没有意义的

#### 测试与调试提示 13.4

在执行 `synchronized` 方法时，所进行的锁定如果永远不打开，就会导致死锁。当异常发生时，Java 的异常机制将和同步机制协调使用，打开相应的同步锁，以避免这种死锁

监控器对象将所有等待进入该对象执行 `synchronized` 方法的线程进行排队。如果一个线程要调用某个对象的 `synchronized` 方法，而此时另一个线程已经在执行该对象的 `synchronized` 方法，那么该线程就入队，等待该对象。如果一个线程在对象中进行操作时调用了 `wait` 方法，那么该线程也要入队。然而，区分开由于监控器忙而阻塞的等待线程以及由于在监控器中显式地调用 `wait` 方法而等待的线程是很重要的。在执行完 `synchronized` 方法后，由于监控器忙而阻塞的外部线程就可以进入该对象继续执行；而那些显式地调用了 `wait` 方法的线程，则只能在其他线程使用 `notify` 或 `notifyAll` 方法将其唤醒时才能继续执行。当一个入队线程具有了继续执行的资格时，调度程序就选择优先级最高的线程来执行。

#### 常见编程错误 13.3

如果一个在监控器外部执行的线程调用了 `wait`、`notify` 或 `notifyAll` 方法，则会产生错误。这会引发一个 `IllegalMonitorStateException` 异常。

## 13.6 未使用线程同步的生产者/消费者关系

在一个生产者/消费者关系中，调用“生产”方法的生产者线程可能会发现，消费者线程还没有从内存的共享区域（称为缓冲区）中读入最新的信息，因此生产者将调用 `wait` 方法。当消费者线程读入信息后，就调用 `notify` 方法通知正在等待的生产者继续执行。当消费者线程进入监控器并发现缓冲区为空时，就调用 `wait` 方法。当生产者发现缓冲区为空时，就向缓冲区中写入信息，然后调用 `notify` 方法，这样，正在等待的消费者就可以继续执行。

如果在多个线程之间不采用同步访问，那么共享数据就可能受到破坏。让我们看一个简单的生产者/消费者关系，生产者线程向一段共享内存中放入一系列数字（我们将使用 0, 1, 2, ...），消费者线程则从该共享内存中读出这些数据，并打印出来。我们将打印生产者所生产的“东西”以及消费者所消费的“东西”。图 13.4 中的程序说明了生产者和消费者如何在不使用同步操作的情况下访问一个共享内存单元。由于这些线程没有同步，因此如果在消费者访问上一个数据之前，生产者就将新的数据放入了共享区中，那么可能导致数据丢失；如果在生产者生产下一个新数据项之前，消费者又访问了一次数据，那么就可能导致数据“重复”。



```

1  // Fig. 13.4: SharedCell.java
2  // Show multiple threads modifying shared object.
3
4  public class SharedCell {
5      public static void main( String args[ ] )
6      {
7          HoldInteger h = new HoldInteger();
8          ProduceInteger p = new ProduceInteger( h );
9          ConsumeInteger c = new ConsumeInteger( h );
10
11          p.start();
12          c.start();
13      }
14  }
15
16  class ProduceInteger extends Thread {
17      private HoldInteger pHold;
18
19      public ProduceInteger( HoldInteger h )
20      {
21          pHold = h;
22      }
23
24      public void run()
25      {
26          for ( int count = 0; count < 10; count++ ) {
27              pHold.setSharedInt( count );
28              System.out.println( "Producer set sharedInt to " +
29                                  count );
30
31              // sleep for a random interval
32              try {
33                  sleep( (int) ( Math.random() * 3000 ) );
34              }
35              catch( InterruptedException e ) {
36                  System.err.println( "Exception " + e.toString() );
37              }
38          }
39      }
40  }
41
42  class ConsumeInteger extends Thread {
43      private HoldInteger cHold;
44
45      public ConsumeInteger( HoldInteger h )
46      {
47          cHold = h;
48      }
49
50      public void run()
51      {
52          int val;
53
54          val = cHold.getSharedInt();
55          System.out.println( "Consumer retrieved " + val );

```

```

56
57     while ( val != 9 ) {
58         // sleep for a random interval
59         try {
60             sleep( (int) ( Math.random() * 3000 ) );
61         }
62         catch( InterruptedException e ) {
63             System.err.println( "Exception " + e.toString() );
64         }
65
66         val = cHold.getSharedInt();
67         System.out.println( "Consumer retrieved " + val );
68     }
69 }
70
71
72 class HoldInteger {
73     private int sharedInt;
74
75     public void setSharedInt( int val ) { sharedInt = val; }
76
77     public int getSharedInt() { return sharedInt; }
78 }

```

**输出结果：**

```

Producer  set sharedInt to 0
Consumer  retrieved 0
Producer  set sharedInt to 1
Consumer  retrieved 1
Producer  set sharedInt to 2
Producer  set sharedInt to 3
Consumer  retrieved 3
Producer  set sharedInt to 4
Producer  set sharedInt to 5
Consumer  retrieved 5
Consumer  retrieved 5
Producer  set sharedInt to 6
Consumer  retrieved 6
Consumer  retrieved 6
Producer  set sharedInt to 7
Producer  set sharedInt to 8
Producer  set sharedInt to 9
Consumer  retrieved 9

```

图 13.4 未使用同步的多线程修改一个共享对象

这个程序包括4个类——SharedCell、ProduceInteger、ConsumeInteger和HoldInteger。在SharedCell类的main方法中（第5行），创建了一个共享的HoldInteger对象h，并将它作为参数传给构造函数，生成ProduceInteger对象p和ConsumeInteger对象c。然后main方法分别调用ProduceInteger对象p及ConsumeInteger对象c的start方法，使它们进入就绪状态。这部分操作主要是启动这些线程。

从Thread类继承的ProduceInteger类包含一个实例变量pHold、一个构造函数及一个run方法。变量pHold在构造函数中初始化（第21行），初值等于参数HoldInteger对象h的值。ProduceInteger类的run方法（第24行）包含一个for循环结构（共循环10次），每次循环都调用HoldInteger类的

setSharedInt方法,并用count的值来设置共享对象的实例变量sharedInt的值。接着,将传入setSharedInt方法的参数值显示出来,即显示count值。然后调用sleep方法,使ProduceInteger对象进入睡眠状态,睡眠时间为0~3秒之间的一个随机值。

ConsumeInteger类(从Thread类继承)包含一个实例变量cHold、一个构造函数及一个run方法。变量cHold在构造函数中初始化(第47行),初值等于参数HoldInteger对象h的值。ConsumeInteger的run方法(第50行)中包含一个while循环结构,它的循环结束条件是从共享内存区中读入值9。在每次循环中,都调用sleep方法,使ConsumeInteger对象进入睡眠状态,睡眠时间为0~3秒之间的一个随机值。然后,再调用HoldInteger类的getSharedInt方法,获取共享对象实例变量ShareInt的值。最后,显示getSharedInt所返回的值。

HoldInteger类的setSharedInt方法(第75行)和getSharedInt方法(第77行)并没有对实例变量SharedInt进行同步访问。在理想情况下,我们希望由ProduceInteger对象p所生产的每一个值,都刚好被ConsumeInteger对象c消费一次。然而,当我们研究一下图13.4的输出结果时就会发现,实际上丢失了值2、4、7和8(也就是说消费者根本没有看到这些值),而值5和6则被消费者错误地重复了一遍(即取回两次)。这个例子清楚地说明了在访问共享数据时,必须仔细控制并发线程,否则程序就会产生错误的结果。

为了解决前面例子中的数据丢失和数据重复问题,我们将使并发的生产者和消费者线程对共享数据的访问同步。使用synchronized关键字声明生产者或消费者所使用的每一个访问共享数据的方法。当某个对象中一个声明为synchronized的方法正在运行时,该对象就被锁定,这时,该对象中的任何其他synchronized方法都无法同时运行。

## 13.7 使用线程同步的生产者/消费者关系

图13.5中的应用程序说明了生产者和消费者如何使用线程同步来访问内存的共享区,以便消费者只有在生产者产生了一个值之后才使用该值。本例中的HoldInteger(第74行)类已经过调整,它现在包含两个实例变量shareInt和writeable,并将setSharedInt方法和getSharedInt方法声明为synchronized方法。因为HoldInteger类中包含了两个synchronized方法,所以HoldInteger类的对象就是监控器。setSharedInt方法可以使用实例变量writeable(即所谓的监控器条件变量),用于确定是否能向共享内存区中写入数据,writeable也可以由getSharedInt方法使用,用于确定是否能从共享内存区中读出数据。

```
1 // Fig. 13.5: SharedCell.java
2 // Show multiple threads modifying shared object.
3 // Use synchronization to ensure that both threads
4 // access the shared cell properly.
5
6 public class SharedCell {
7     public static void main( String args[ ] )
8     {
9         HoldInteger h = new HoldInteger();
10        ProduceInteger p = new ProduceInteger( h );
11        ConsumeInteger c = new ConsumeInteger( h );
12
13        p.start();
14        c.start();
15    }
```

```
16     )
17
18     class ProduceInteger extends Thread {
19         private HoldInteger pHold;
20
21         public ProduceInteger( HoldInteger h )
22         {
23             pHold = h;
24         }
25
26         public void run()
27         {
28             for ( int count = 0; count < 10; count++ ) {
29                 pHold.setSharedInt( count );
30                 System.out.println( "Producer set sharedInt to " +
31                                     count );
32
33                 // sleep for a random interval
34                 try {
35                     sleep( (int) ( Math.random() * 3000 ) );
36                 }
37                 catch( InterruptedException e ) {
38                     System.err.println( "Exception " + e.toString() );
39                 }
40             }
41         }
42     }
43
44     class ConsumeInteger extends Thread {
45         private HoldInteger cHold;
46
47         public ConsumeInteger( HoldInteger h )
48         {
49             cHold = h;
50         }
51
52         public void run()
53         {
54             int val;
55
56             val = cHold.getSharedInt();
57             System.out.println( "Consumer retrieved " + val );
58
59             while ( val != 9 ) {
60                 // sleep for a random interval
61                 try {
62                     sleep( (int) ( Math.random() * 3000 ) );
63                 }
64                 catch( InterruptedException e ) {
65                     System.err.println( "Exception " + e.toString() );
66                 }
67
68                 val = cHold.getSharedInt();
69                 System.out.println( "Consumer retrieved " + val );
70             }
71         }
72     }
```

```

72     }
73
74     class HoldInteger {
75         private int sharedInt;
76         private boolean writeable = true;
77
78         public synchronized void setSharedInt( int val )
79         {
80             while ( !writeable ) {
81                 try {
82                     wait();
83                 }
84                 catch ( InterruptedException e ) {
85                     System.err.println( "Exception: " + e.toString() );
86                 }
87             }
88
89             sharedInt = val;
90             writeable = false;
91             notify();
92         }
93
94         public synchronized int getSharedInt()
95         {
96             while ( writeable ) {
97                 try {
98                     wait();
99                 }
100                catch ( InterruptedException e ) {
101                    System.err.println( "Exception: " + e.toString() );
102                }
103            }
104
105            writeable = true;
106            notify();
107            return sharedInt;
108        }
109    }

```

**输出结果:**

```

Producer set sharedInt to 0
Consumer retrieved 0
Producer set sharedInt to 1
Consumer retrieved 1
Producer set sharedInt to 2
Consumer retrieved 2
Producer set sharedInt to 3
Consumer retrieved 3
Producer set sharedInt to 4
Consumer retrieved 4
Producer set sharedInt to 5
Consumer retrieved 5
Producer set sharedInt to 6
Consumer retrieved 6
Producer set sharedInt to 7
Consumer retrieved 7

```

```
Producer  set sharedInt to 8
Consumer  retrieved 8
Producer  set sharedInt to 9
Consumer  retrieved 9
```

图 13.5 使用同步的多线程修改一个共享对象

实例变量 `writable` 是一个布尔变量, `HoldInteger` 类的两个方法都要使用它。如果 `writable` 的值为 `true`, 那么 `setSharedInt` 就可以向变量 `sharedInt` 中写入一个值, 因为此时该变量没有任何值。同时, 这也意味着此时 `getSharedInt` 方法不能从 `sharedInt` 中读入值。如果 `writable` 为 `false`, 那么 `getSharedInt` 方法就可以从变量 `sharedInt` 中读入一个值, 因为此时该变量有值。同时, 这也意味着此时 `setSharedInt` 方法不能向 `sharedInt` 中写入值。

当 `ProduceInteger` 对象调用 `setSharedInt` 方法时, `HoldInteger` 对象就被锁定。第 80 行是一个 `while` 结构, 利用条件 “`!writable`” 来判断 `writable` 的值。如果该条件为 `true`, 就调用 `wait` 方法。这是将调用 `setSharedInt` 方法的 `ProduceInteger` 对象放到 `HoldInteger` 对象的等待队列中, 并将该 `HoldInteger` 对象的锁打开, 使得可以调用该对象的其他 `synchronized` 方法。这个 `ProduceInteger` 对象将一直放在等待队列中, 直到接到通知可以继续执行。这时, 该对象就进入了就绪状态, 等待分配处理器。当这个 `ProduceInteger` 对象再次进入运行状态时, 就将隐含地锁定 `HoldInteger` 对象, 而 `getSharedInt` 方法将继续执行 `while` 结构中 `wait` 后面的语句。在本例中, `wait` 后面没有其他语句, 因此将进行下一次循环, 判断 `while` 条件。如果该条件值为 `false`, 那么就将 `sharedInt` 赋值为 `val` (`val` 是 `setSharedInt` 的参数), 将 `writable` 设置为 `false`, 表示共享区已有值, 然后调用 `notify` 方法。如果此时存在等待线程, 那么等待队列中的第一个线程就进入就绪状态, 表示该线程现在可以继续执行了 (只要分配了处理器就可执行)。执行完后, `notify` 方法立即返回, 而 `setSharedInt` 方法则返回给它的调用者。

`getSharedInt` 方法和 `setSharedInt` 方法在实现上是类似的。当 `ConsumeInteger` 对象调用 `getSharedInt` 方法时, `HoldInteger` 对象就被锁定。第 96 行是一个 `while` 结构, 判断 `writable` 变量的值。如果 `writable` 的值为 `true` (也就是没有任何数据可用), 那么就调用 `wait` 方法。这是将调用 `getSharedInt` 方法的 `ConsumeInteger` 对象放到 `HoldInteger` 对象的等待队列中, 并将该 `HoldInteger` 对象的锁打开, 使得可以调用该对象的其他 `synchronized` 方法。这个 `ConsumeInteger` 对象将一直放在等待队列中, 直到通知它可以继续执行——这时, 该对象将进入就绪状态, 等待分配处理器。当 `ConsumeInteger` 对象再次进入运行状态时, `HoldInteger` 对象就被隐含地锁定, 而 `getSharedInt` 方法将继续执行 `while` 结构中 `wait` 后面的语句。本例子中, `wait` 后面没有其他语句, 因此再一次循环, 判断 `while` 条件。如果该条件值为 `false`, 那么就将 `writable` 设置为 `true`, 表示共享区已空, 然后调用 `notify` 方法。如果此时存在等待线程, 那么等待队列中的第一个线程就进入就绪状态, 表示该线程现在可以继续执行 (只要分配了处理器就可执行)。`notify` 方法立即返回, 而 `sharedInt` 的值则返回它的调用者。

图 13.5 的输出结果表明, 每个“生产”的整数都只“消费”了一次——没有出现数据丢失的情况, 也没有重复使用数据。

## 13.8 生产者/消费者的关系: 循环缓冲区

图 13.5 的程序确实能够正确地访问共享数据, 但它的运行性能却不是很好。因为这些线程都在异步运行, 因此我们无法预测它们的相对速度。如果生产者的生产速度要想比消费者的消费速度快一些, 那么它是无法做到的。为了使生产者能够连续不断地进行生产, 可以使用一个循环缓冲区,

这个缓冲区中有足够的附加单元来处理“额外”的产品。图13.6的程序说明了一个生产者和一个消费者如何使用同步操作来访问一个循环缓冲区（这是一个有5个单元的数组）。当缓冲区中有一个或多个值时，消费者只能消费一个值；当缓冲区有一个或多个可用的空单元时，生产者也只能生产一个值。我们将该程序设计成一个applet，所产生的输出结果将发送到一个文本区域中。SharedCell类的start方法创建了一个HoldInteger对象、一个ProduceInteger对象及一个ConsumeInteger对象。其中，HoldInteger对象h的参数是一个TextArea对象，该程序的输出结果将在这个文本区域中显示。

```

1  // Fig. 13.6: SharedCell.java
2  // Show multiple threads modifying shared object.
3  // Use synchronization to ensure that both threads
4  // access the shared cell properly.
5  import java.awt.*;
6  import java.applet.Applet;
7
8  public class SharedCell extends Applet {
9      private TextArea output;
10
11      public void init()
12      {
13          output = new TextArea( 28, 48 );
14          add( output );
15      }
16
17      public void start()
18      {
19          HoldInteger h = new HoldInteger( output );
20          ProduceInteger p = new ProduceInteger( h );
21          ConsumeInteger c = new ConsumeInteger( h );
22
23          p.start();
24          c.start();
25      }
26  }
27
28  class ProduceInteger extends Thread {
29      private HoldInteger pHold;
30      private TextArea output;
31
32      public ProduceInteger( HoldInteger h )
33      {
34          pHold = h;
35      }
36
37      public void run()
38      {
39          for ( int count = 0; count < 10; count++ ) {
40              pHold.setSharedInt( count );
41
42              // sleep for a random interval
43              try {
44                  sleep( (int) ( Math.random() * 500 ) );
45              }
46              catch( InterruptedException e ) {
47                  System.err.println( "Exception " + e.toString() );
48              }
49          }
50      }
51  }

```

```
50     }
51 }
52
53 class ConsumeInteger extends Thread {
54     private HoldInteger cHold;
55
56     public ConsumeInteger( HoldInteger h )
57     {
58         cHold = h;
59     }
60
61     public void run()
62     {
63         int val;
64
65         val = cHold.getSharedInt();
66
67         while ( val != 9 ) {
68             // sleep for a random interval
69             try {
70                 sleep( (int) ( Math.random() * 3000 ) );
71             }
72             catch( InterruptedException e ) {
73                 System.err.println( "Exception " + e.toString() );
74             }
75
76             val = cHold.getSharedInt();
77         }
78     }
79 }
80
81 class HoldInteger {
82     private int sharedInt[] = { 9, 9, 9, 9, 9 };
83     private boolean writeable = true;
84     private boolean readable = false;
85     private int readLoc = 0, writeLoc = 0;
86     private TextArea output;
87
88     public HoldInteger( TextArea out )
89     {
90         output = out;
91     }
92
93     public synchronized void setSharedInt( int val )
94     {
95         while ( !writeable ) {
96             try {
97                 output.appendText( " WAITING TO PRODUCE " + val );
98                 wait();
99             }
100             catch ( InterruptedException e ) {
101                 System.err.println( "Exception: " + e.toString() );
102             }
103         }
104
105         sharedInt[ writeLoc ] = val;
106         readable = true;
107
108         output.appendText( " \nProduced " + val +
```



```

109         " into cell " + writeLoc );
110
111     writeLoc = ++writeLoc % 5;
112
113     output.appendText( " \twrite " + writeLoc +
114                       " \tread " + readLoc );
115     printBuffer( output, sharedInt );
116
117     if ( writeLoc == readLoc )
118         writeable = false;
119         output.appendText( " \nBUFFER FULL" );
120     }
121
122     notify();
123 }
124
125 public synchronized int getSharedInt()
126 {
127     int val;
128
129     while ( !readable ) {
130         try {
131             output.appendText( " WAITING TO CONSUME" );
132             wait();
133         }
134         catch ( InterruptedException e ) {
135             System.err.println( "Exception: " + e.toString() );
136         }
137     }
138
139     writeable = true;
140     val = sharedInt[ readLoc ];
141
142     output.appendText( " \nConsumed " + val +
143                       " from cell " + readLoc );
144
145     readLoc = ++readLoc % 5;
146
147     output.appendText( " \twrite " + writeLoc +
148                       " \tread " + readLoc );
149     printBuffer( output, sharedInt );
150
151     if ( readLoc == writeLoc ) {
152         readable = false;
153         output.appendText( " \nBUFFER EMPTY" );
154     }
155
156     notify();
157     return val;
158 }
159
160 public void printBuffer( TextArea out, int buf[ ] )
161 {
162     output.appendText( " \tbuffer: " );
163
164     for ( int i = 0; i < buf.length; i++ )
165         out.appendText( " " + buf[ i ] );
166 }
167 }

```

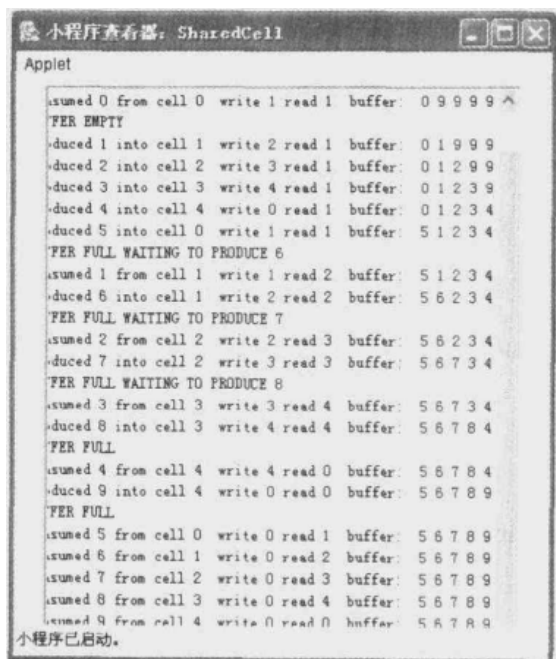


图 13.6 使用同步的多线程来修改共享对象——一个具有多个单元的数组

在这个例子中，主要的修改还是集中在对 `HoldInteger` 类的定义。现在，`HoldInteger` 类中包含 6 个实例变量——`sharedInt` 是一个有 5 个元素的整型数组，作为循环缓冲区；`writable` 用于指示生产者是否可以向循环缓冲区中写入数据；`readable` 用于指示消费者是否可以从小循环缓冲区读出数据；`readLoc` 用于表示消费者下一个可读取值的当前位置；`writeLoc` 用于表示生产者下一个可写入值的位置；`output` 是一个用于显示输出结果的文本区域。

`setSharedInt` 方法的功能与图 13.5 的程序类似，只做了一点点改动。当程序继续执行 `while` 循环后面的第 105 行语句时，由生产者生产的值将放入循环缓冲区的 `writeLoc` 位置上。然后，将 `readable` 设置成 `true`，因为此时缓冲区中至少已有一个值可以读出。所生产的值以及该值所在的位置都通过 `appendText` 方法而添加到文本区域中。然后，更新 `writeLoc` 的值，以备下一次调用 `setSharedInt`。接着，在文本区域中继续输出 `writeLoc` 和 `readLoc` 的当前值以及循环缓冲区中的当前值。如果 `writeLoc` 与 `readLoc` 的值相等，那么表示循环缓冲区已满，因此将 `writable` 设置为 `false`，并显示字符串 `BUFFER FULL`。最后，调用 `notify` 方法，使下一个等待线程进入就绪状态。

`getSharedInt` 方法的功能也和图 13.5 的程序类似，也只做了一点点改动。当程序继续执行 `while` 循环后的第 140 行时，`writable` 的值将设置成 `true`，因为在缓冲区中至少已有一个空位置可以写入数据。然后，将循环缓冲区中位于 `readLoc` 位置上的值赋给 `val`。接着，使用 `appendText` 方法将 `val` 的值及其所在的位置 `readLoc` 添加到文本区域中。然后更新 `readLoc` 的值，以备下一次调用 `getSharedInt`。接着，在文本区域中继续输出 `writeLoc` 和 `readLoc` 的当前值以及循环缓冲区中的当前值。如果 `readLoc` 与 `writeLoc` 的值相等，那么表示循环缓冲区已空，因此将 `readable` 设置为 `false`，并显示字符串 `BUFFER EMPTY`。最后，调用 `notify` 方法，使下一个等待线程进入就绪状态。

在图 13.6 的程序中，输出结果已经增多，其中包含了 `writeLoc` 和 `readLoc` 的当前值。另外，图中还显示了缓冲区 `sharedInt` 的当前内容。`sharedInt` 数组的每个元素都将初始化为 9，这样，在输出结果中就可以看出哪一个值是新插入的。注意，当把第 5 个值放入缓冲区的第 5 个元素中后，第 6 个值将放在缓冲区的第 1 个位置上——这就是循环缓冲区的循环功能。所有的输出语句都已从 `ProduceInteger` 类和 `ConsumeInteger` 类中删除。

## 13.9 精灵线程

精灵 (daemon) 线程是为其他线程服务的一种线程。精灵线程都在后台运行 (即当处理器有空闲时就运行精灵线程, 以免浪费处理器时间)。无用单元回收处理程序就是一个精灵线程。非精灵线程就是传统的用户线程。我们可以调用下面的方法, 将一个线程指定为精灵线程:

```
setDaemon (true );
```

如果该方法的参数值为 false, 则表示该线程不是精灵线程。在一个程序中, 可以同时包含精灵线程和非精灵线程。当程序中只剩下精灵线程时, Java 就会退出。如果想使某个线程成为精灵线程, 那么必须在调用 start 方法之前进行设置, 否则将会引发 `IllegalThreadStateException` 异常。如果一个线程为精灵线程, 那么 `isDaemon` 方法将返回 true 值, 否则就返回 false 值。

## 13.10 Runnable 接口

C++ 支持多继承, 即一个子类可以继承多个超类。多继承是一个功能强大的属性, 但是它使用起来比较复杂, 容易引起混淆及性能问题。而 Java 则不支持多继承, 就像 Java 也取消了很多其他复杂的 C++ 特性一样。但是, Java 支持接口的概念, 这是一种更简单的方案, 而且能够提供多继承的很多主要优点。

在此之前, 我们都是通过扩展 Thread 类来创建新类, 以支持多线程。我们曾经重写过 run 方法来指定需要并发完成的任务。

采用这种方案的一个问题是其创建了一些相当奇怪的关系, 因为使用 extend 实际上是在完成继承操作, 它表示一个子类对象 “是” 其超类的一个对象。

如果我们想要用从其他类 (而不是 Thread 类) 中派生的类来支持多线程, 那么可以在该类中提供 Runnable 接口。Thread 类本身就提供了 Runnable 接口, 如下所示:

```
public class Thread extends Object implements Runnable
```

Runnable 接口使我们可以将新类看成是一个 Runnable 对象 (这就像使用了类继承后, 可以将子类看成是其超类的一个对象)。如同从 Thread 类派生一样, 控制线程的代码仍然放在 run 方法中。

我们可以利用下面的 Thread 类的构造函数来创建一个线程, 传入的参数是一个对象引用, 该对象是提供了 Runnable 接口的类的对象:

```
public Thread ( Runnable runnableObject)
```

这个 Thread 构造函数将 runnableObject 的 run 方法标记为该线程开始执行时将要引用的方法。

下面的构造函数是创建一个名称为 threadName 的线程, 并将第一个参数 runnableObject 的 run 方法标记为该线程开始执行时将要引用的方法:

```
public Thread ( Runnable runnableObject ,String threadName)
```

图 13.7 的程序是一个提供了 Runnable 接口的 applet。applet 类 RandomCharacters 显示了三个文本字段和三个按钮。每对文本字段和按钮都有一个与之相关的独立线程。该 applet 还定义了一个字符串变量 alphabet, 它的值为字母 A 到 Z。这个字符串将由三个线程所共享。在 applet 的 start 方法中 (第 38 行) 创建了三个 Thread 对象, 并都初始化为 this 类型 (即该 applet 对象)。然后调用每个线程的 start 方法, 使它们都进入就绪状态。

```
1 // Fig. 13.7: RandomCharacters.java
2 // Demonstrating the Runnable interface
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class RandomCharacters extends Applet
7         implements Runnable {
8
9     String alphabet;
10    TextField output1, output2, output3;
11    Button button1, button2, button3;
12
13    Thread thread1, thread2, thread3;
14
15    boolean suspend1, suspend2, suspend3;
16
17    public void init()
18    {
19        alphabet = new String( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
20        output1 = new TextField( 10 );
21        output1.setEditable( false );
22        output2 = new TextField( 10 );
23        output2.setEditable( false );
24        output3 = new TextField( 10 );
25        output3.setEditable( false );
26        button1 = new Button( "Suspend/Resume 1" );
27        button2 = new Button( "Suspend/Resume 2" );
28        button3 = new Button( "Suspend/Resume 3" );
29
30        add( output1 );
31        add( button1 );
32        add( output2 );
33        add( button2 );
34        add( output3 );
35        add( button3 );
36    }
37
38    public void start()
39    {
40        // create threads and start every time start is called
41        thread1 = new Thread( this, "Thread 1" );
42        thread2 = new Thread( this, "Thread 2" );
43        thread3 = new Thread( this, "Thread 3" );
44
45        thread1.start();
46        thread2.start();
47        thread3.start();
48    }
49
50    public void stop()
51    {
52        // stop threads every time stop is called
53        // as the user browses another Web page
54        thread1.stop();
55        thread2.stop();
56        thread3.stop();
57    }
58 }
```

```
57     }
58
59     public boolean action( Event event, Object obj )
60     {
61         if ( event.target == button1 )
62             if ( suspend1 ) {
63                 thread1.resume();
64                 suspend1 = false;
65             }
66             else {
67                 thread1.suspend();
68                 output1.setText( "suspended" );
69                 suspend1 = true;
70             }
71         else if ( event.target == button2 )
72             if ( suspend2 ) {
73                 thread2.resume();
74                 suspend2 = false;
75             }
76             else {
77                 thread2.suspend();
78                 output2.setText( "suspended" );
79                 suspend2 = true;
80             }
81         else if ( event.target == button3 )
82             if ( suspend3 ) {
83                 thread3.resume();
84                 suspend3 = false;
85             }
86             else {
87                 thread3.suspend();
88                 output3.setText( "suspended" );
89                 suspend3 = true;
90             }
91
92         return true;
93     }
94
95     public void run()
96     {
97         int location;
98         char display;
99         String executingThread;
100
101         while ( true ) {
102             // sleep from 0 to 5 seconds
103             try {
104                 Thread.sleep( (int) ( Math.random() * 5000 ) );
105             }
106             catch ( InterruptedException e ) {
107                 e.printStackTrace();
108             }
109
110             location = (int) ( Math.random() * 26 );
111             display = alphabet.charAt( location );
112
```

```

113         executingThread = Thread.currentThread().getName();
114
115         if ( executingThread.equals( "Thread 1" ) )
116             output1.setText( "Thread 1: " + display );
117         else if ( executingThread.equals( "Thread 2" ) )
118             output2.setText( "Thread 2: " + display );
119         else if ( executingThread.equals( "Thread 3" ) )
120             output3.setText( "Thread 3: " + display );
121     }
122 }
123 }

```

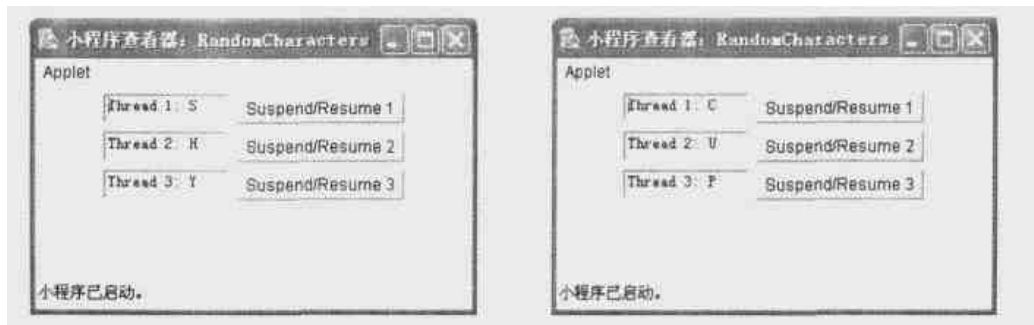


图 13.7 Runnable 接口

在 run 方法中（第 95 行）包含一个无限循环。每循环一次，该线程都睡眠一个随机的时间间隔（0~5 秒之间）。然后，再从 alphabet 字符串中随机选择一个字符。下面的语句首先利用 Thread.currentThread 方法来确定当前执行的是哪个 Thread 对象，然后再使用 getName 方法获得它的名称：

```
executingThread = Thread.currentThread ( ).getName ( );
```

从 getName 中获得的字符串将在嵌套的 if/else 结构中用于选择显示字符的文本字段。

如果用户使用鼠标点击了某个文本字段右边的 Suspend/Resume 按钮，那么就将调用 action 方法（第 59 行）。action 方法首先判断产生事件的是哪个按钮，然后再通过判断布尔变量 suspend1、suspend2 或 suspend3（每个线程对应一个变量）的值，确定相应的线程是否已被挂起。如果相应的布尔变量值为 true，那么就调用该线程的 resume 方法，并将布尔变量的值设置为 false。否则，就调用该线程的 suspend 方法，并在该线程所对应的文本字段中显示字符串“suspended”，然后将布尔变量的值设置为 true。

该 applet 还提供了一个 stop 方法，如果用户离开了该 applet 所在的网页，那么可以使用 stop 方法将这三个线程都停止。如果用户又返回了该网页，那么这三个线程将重新创建，因为一旦再次访问该网页，就会调用 applet 的 start 方法。

#### 性能提示 13.7

离开网页时应将 applet 线程停止，这是一个礼貌性的编程习惯，可以防止出现 applet 根本未被浏览，但却占用浏览器所在机器的处理器时间（这会降低性能）的情况。这些线程可以通过 applet 的 start 方法重新启动。当用户再次访问网页时，将自动调用 start 方法。

## 13.11 线程组

有时，将一些线程标识为属于同一个线程组是很有用的，ThreadGroup 类中就包含一些用于创

建和操作线程组的方法。在构造函数中,可以通过一个String参数给线程组设置一个惟一的名称。

一个线程组中的各个线程可以作为一个组进行处理。例如,可以挂起一组中的所有线程,或恢复一组中的所有线程,甚至可以删除一组中的所有线程。一个线程组可以成为一个子线程组的父线程组。发送给一个父线程组的方法也可以发送给它的子线程组的所有线程。

ThreadGroup类提供了两个构造函数。下面的构造函数用于创建一个名为stringName的ThreadGroup:

```
public ThreadGroup ( String stringName
```

下面的构造函数用于创建线程组parentThreadGroup的一个子线程组,名称为stringName:

```
public ThreadGroup (ThreadGroup parentThreadGroup, String stringName )
```

Thread类提供了一个构造函数,用于创建Thread,并将其与一个ThreadGroup联系起来。下面的构造函数用于创建一个属于线程组threadGroup的线程,该线程的名称为stringName:

```
public Thread (ThreadGroup threadGroup, String stringName )
```

这个构造函数通常是由Thread的派生类调用的,该派生类的对象应与一个线程组联系起来。

下面的构造函数用于创建一个属于线程组threadGroup的线程,当该线程分配到一个处理器而开始执行时,将会调用RunnableObject的run方法:

```
public Thread (ThreadGroup threadGroup, Runnable runnableObject )
```

下面的构造函数用于创建一个属于线程组threadGroup的线程,当该线程分配到一个处理器而开始执行时,将会调用RunnableObject的run方法。该线程的名称为stringName:

```
public Thread (ThreadGroup threadGroup, Runnable runnableObject,  
                String stringName)
```

ThreadGroup类中包含很多用于处理线程组的方法。下面列出了一部分方法。如果了解有关这些方法的更多信息,可参见Java API的资料。

1. activeCount 方法用于报告一个线程组及其所有子线程组中的活动线程的数目。
2. enumerate 方法有四个版本。其中两个版本是将ThreadGroup中的活动线程复制到一个Thread数组中(其中一个版本还可以递归地复制子线程组中的所有活动线程)。另外两个版本是将ThreadGroup中的活动子线程组复制到一个Thread数组中(其中一个版本还可以递归地复制所有子线程组的所有活动线程组)。
3. getMaxPriority 方法用于返回一个线程组的最大优先级。  
setMaxPriority 方法用于为一个线程组设置新的最大优先级。
4. getName 方法用于获取线程组的名称。
5. getParent 方法用于获取一个线程组的父线程组。
6. 如果要判断的线程组是参数ThreadGroup的父线程组或是同一个线程组,那么parentOf方法将返回true,否则将返回false。
7. stop 方法用于停止线程组及其所有子线程组中的每个线程。
8. suspend 方法用于挂起线程组及其所有子线程组中的每个线程。
9. resume 方法用于恢复线程组及其所有子线程组中的每个线程。
10. destroy 方法用于删除一个线程组及其子线程组。

## 测试与调试提示 13.5

list 方法用于列出线程组的内容，这样有助于程序调试。

## 小结

- 计算机可以并发地完成很多操作，比如编译程序、打印文件，并从网络上接收电子邮件。
- 程序语言一般只提供几种简单的控制结构，利用这些控制结构，一次只能执行一个动作。只有前一个动作完成后，才能开始执行下一个动作。
- 目前，计算机上的并发操作通常都是利用操作系统的“原语操作”来实现的，而这些“原语操作”只有那些经验丰富的“系统程序员”才能使用。
- Java 为程序员提供了并发的原语操作。
- 应用程序中可以包含线程，每个线程都完成程序的某一部分功能，并且可以与其他线程并发执行，这种机制称为多线程。
- Java 提供了一个低优先级的无用单元回收处理线程，它可以自动动态回收不再需要的内存空间。当处理器空闲并且没有其他高优先级的线程运行时，就运行无用单元回收处理程序。不过，当系统发生内存溢出时，无用单元回收处理程序会立即启动来回收内存。
- run 方法中包含控制线程执行的代码。
- 在一个程序中，通过调用 start 方法（然后再调用 run 方法）可以启动一个新线程。
- interrupt 方法用于中断一个线程。如果当前线程已经中断，那么 interrupt 方法将返回 true 值，否则将返回 false 值。isInterrupted 方法用于确定一个线程是否已经中断。
- stop 方法用于停止一个线程，并引发一个 ThreadDeath 对象；ThreadDeath 是 Error 的一个子类。
- 如果某个线程已调用了 start，但还没有调用 stop，那么 isAlive 方法返回 true 值。
- setName 方法用于设置 Thread 的名称。getName 方法用于返回一个 String，其中包括线程的名称、线程的优先级以及线程所在的线程组。
- currentThread 方法用于返回对当前 Thread 的一个引用。
- join 方法用于等待目标线程接收终止信息，以便当前线程可以执行。
- 等待是很危险的；它可能导致两个特别严重的问题：死锁和无限延迟；无限延迟也称为互斥等待。
- 一个线程在刚刚创建时称为“新生”状态。该线程在调用 start 方法之前一直处于“新生”状态；在调用 start 方法之后，该线程进入“就绪”状态。
- 当系统为线程分配了处理器时，具有最高优先级的就绪线程将进入“运行”状态。
- 当一个线程的 run 方法运行完毕，或调用了 stop 方法时，该线程就进入“停止”状态；一个停止的线程将被系统永远删除。
- 当一个正在运行的线程发出一个输入/输出请求时，就会进入“阻塞”状态。当输入/输出请求得到回应时，“阻塞”线程就会进入“就绪”状态。一个阻塞线程无法直接使用处理器（即使处理器空闲也无法使用）。
- 当调用一个运行线程的 sleep 方法时，该线程就进入“睡眠”状态。一个睡眠的线程在超过了睡眠时间之后就进入“就绪”状态。一个睡眠线程无法直接使用处理器（即使处理器空闲也无法使用）。
- 当一个运行线程调用 suspend 方法时，该线程就进入“挂起”状态。当一个“挂起”线程的



resume 方法由另一个线程调用时,该线程就进入就绪状态(挂起线程不能直接使用处理器(即使处理器空闲也无法使用))

- 当一个运行线程调用 wait 方法时,就进入“等待”状态,即进入一个与调用 wait 的对象相关的等待队列中进行等待。当另一个与该对象相关的线程发出 notify 调用时,等待队列中的第一个线程将进入“就绪”状态。
- 当一个与某个对象相关的线程发出 notifyAll 调用时,该对象的等待队列中的每个线程都将进入“就绪”状态。
- 每个 Java 线程都有一个优先级,范围在 Thread.MIN\_PRIORITY (该常量的值为 1) 和 Thread.MAX\_PRIORITY (该常量的值为 10) 之间。默认情况下,每个线程的优先级都为 Thread.NORM\_PRIORITY (该常量的值为 5)。
- 有些 Java 平台支持时间分片方式,有些则不支持。如果不采用时间分片方式,那么具有相同优先级的每个线程都会一直执行,直到结束才把处理器让给其他的同级线程执行。如果采用时间分片方式,那么每个线程都只能在一个短暂的处理器时间分片内执行。当时间分片结束时,即使线程还没有执行完,也要将处理器让给下一个具有相同优先级的线程。
- Java 调度程序的功能是保证在任意时刻运行的线程都是优先级最高的。如果采用时间分片方式,那么调度程序就要确保具有相同优先级的每个线程都能以轮转方式轮流运行一个时间分片。
- 一个线程的优先级可以利用 setPriority 方法进行调整, getPriority 方法用于返回线程的优先级。
- 一个线程可以调用 yield 方法来为其他线程提供执行的机会。
- 每个带有 synchronized 方法的对象都是一个监控器。监控器一次只允许一个线程对该对象执行 synchronized 方法。
- 一个执行 synchronized 方法的线程可以判断出自己无法再继续执行,因此自愿地调用 wait 方法。这可以使该线程不再争用处理器和监控器对象。
- 一个已调用了 wait 方法的线程可被另一个调用 notify 方法的线程唤醒。notify 就像是发给等待线程的一个信号,告诉该线程它所等待的条件现在已满足了,因此有资格重新进入监控器。
- 精灵线程是为其他线程服务的。当程序中只剩下精灵线程时,Java 就会退出。如果一个线程想要成为精灵线程,那么它必须在调用 start 方法之前进行设定。
- 如果我们想要用从其他类(而不是 Thread 类)中派生的类来支持多线程,那么可以在该类中提供 Runnable 接口。
- 提供 Runnable 接口使我们可以将新类看成是一个 Runnable 对象(这就像使用了类继承之后,可以将子类看成是其超类的一个对象一样)。如同从 Thread 类派生一样,控制线程的代码仍然放在 run 方法中。
- 使用 Runnable 类的线程可以通过一个 Thread 类的构造函数来创建,所传入的参数是一个对象引用。该对象是提供了 Runnable 接口的类的对象。这个 Thread 构造函数将 runnableObject 的 run 方法标记为该线程开始执行时将要引用的方法。
- ThreadGroup 类中包含一些用于创建和操作线程组的方法。

## 术语

|                                             |                             |                                |                        |
|---------------------------------------------|-----------------------------|--------------------------------|------------------------|
| asynchronous threads                        | 异步线程                        | InterruptedException class     | InterruptedException 类 |
| blocked on I/O                              | I/O 阻塞                      | interrupt method               | interrupt 方法           |
| blocked (state of a thread)                 | 阻塞(线程的一种状态)                 | InterruptedException class     | InterruptedException 类 |
| busy wait                                   | 忙等待                         | interrupted method             | interrupted 方法         |
| child thread group                          | 子线程组                        | interthread communication      | 内部线程通信                 |
| circular buffer                             | 循环缓冲区                       | I/O completion                 | I/O 完成                 |
| concurrency                                 | 并发                          | isAlive method                 | isAlive 方法             |
| concurrent execution of threads             | 线程的并发执行                     | isDaemon method                | isDaemon 方法            |
| condition variable                          | 条件变量                        | isInterrupted method           | isInterrupted 方法       |
| consumer                                    | 消费者                         | join method                    | join 方法                |
| consumer thread                             | 消费者线程                       | kill a thread                  | 删除一个线程                 |
| context                                     | 上下文                         | MAX_PRIORITY(10)               |                        |
| countStackFrames method                     | countStackFrames 方法         | memory leak                    | 内存泄漏                   |
| currentThread method                        | currentThread 方法            | MIN_PRIORITY(1)                |                        |
| daemon thread                               | 精灵线程                        | monitor                        | 监控器                    |
| dead(state of a thread)                     | 停止(线程的一种状态)                 | multiple inheritance           | 多继承                    |
| death of a thread                           | 线程停止                        | multiprocessing                | 多进程                    |
| deadlock                                    | 死锁                          | multithreaded program          | 多线程的程序                 |
| destroy method                              | destroy 方法                  | multithreaded server           | 多线程的服务器                |
| dumpStack method                            | dumpStack 方法                | multithreading                 | 多线程的                   |
| Error class ThreadDeath is a subclass       | ThreadDeath 类是 Error 类的一个子类 | new (state of a thread)        | 线程的新状态                 |
| execution context                           | 执行上下文                       | new Thread                     | 新的 Thread              |
| fixed-priority scheduling                   | 固定优先级调度                     | non-preemptive scheduling      | 非抢占调度                  |
| garbage collection by a low-priority thread | 无用单元回收处理线程(低优先级)            | NORM_PRIORITY(5)               |                        |
| getName method                              | getName 方法                  | notify method                  | notify 方法              |
| getParent method of ThreadGroup class       | ThreadGroup 类的 getParent 方法 | notifyAll method               | notifyAll 方法           |
| getTreadGroup()                             |                             | parallelism                    | 并行性                    |
| highest-priority runnable thread            | 优先级最高的 runnable 线程          | parent thread group            | 父线程组                   |
| IllegalArgumentExpection                    |                             | parent thread                  | 父线程                    |
| IllegalMonitorStateException                |                             | preemptive scheduling          | 抢占调度                   |
| IllegalThreadStateException                 |                             | printStackTrace method         | printStackTrace 方法     |
| indefinite postponement                     | 无限延迟                        | priority of a thread           | 线程的优先级                 |
| inherit thread priority                     | 继承线程的优先级                    | producer                       | 生产者                    |
| init method                                 | init 方法                     | producer/consumer relationship | 生产者/消费者关系              |
| InterruptedException                        |                             | producer thread                | 生产者线程                  |
|                                             |                             | programmer-defined thread      | 程序员自定义的线程              |
|                                             |                             | quantum                        | 时间分片                   |
|                                             |                             | queue in a monitor             | 监控器的队列                 |

- race condition 竞争条件  
 resume method resume 方法  
 resume method of ThreadGroup class ThreadGroup 类的 resume 方法  
 ring buffer 环形缓冲区  
 round-robin scheduling 轮转调度  
 run method run 方法  
 runnable state (of a thread) 一个线程的可运行状态  
 Runnable interface (in java.lang package) Runnable 接口 (在 java.lang 软件包中)  
 running (thread state) 运行 (线程状态)  
 scheduler 调度程序  
 scheduling a thread 调度线程  
 SecurityException  
 set a thread to null 将一个线程设置为空  
 setDaemon method setDaemon 方法  
 setName method setName 方法  
 setPriority method setPriority 方法  
 shared objects 共享对象  
 single-threaded languages 单线程语言  
 single-threaded program 单线程程序  
 sleep method of Thread class Thread类的sleep方法  
 sleeping state (of a thread) 一个线程的睡眠状态  
 start method start 方法  
 starvation 互斥等待  
 stop method of ThreadGroup class ThreadGroup类的 stop 方法  
 suspend method of ThreadGroup class ThreadGroup 类的 suspend 方法  
 synchronization 同步  
 synchronized method synchronized 方法  
 thread 线程  
 Thread class (in java.lang package) Thread类 (在 java.lang 软件包中)  
 ThreadDeath exception ThreadDeath 异常  
 thread group 线程组  
 ThreadGroup class ThreadGroup 类  
 Thread.MAX\_PRIORITY  
 Thread.MIN\_PRIORITY  
 Thread.NORM\_PRIORITY  
 thread priority 线程优先级  
 thread safe 线程安全性  
 Thread.sleep()  
 thread states 线程状态  
 thread synchronization 线程同步  
 timeslicing 时间分片方式  
 wait method wait 方法  
 yield method yield 方法

## 自测练习

### 13.1 填空:

- C 和 C++ 是 \_\_\_\_\_ 线程的语言, 而 Java 则是一个 \_\_\_\_\_ 线程的语言。
- Java 提供了一个可以自动回收动态分配的内存空间的 \_\_\_\_\_ 线程。
- 当程序没有显式地回收动态分配的内存时, 使用 Java 可以消除大部分在 C 和 C++ 这样的语言中通常会发生的 \_\_\_\_\_ 错误。
- \_\_\_\_\_ 方法用于终止一个 Thread 的执行。
- 线程不能运行 (即被阻塞) 的四个原因是 \_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- 线程进入停止状态的两个原因是 \_\_\_\_\_ 和 \_\_\_\_\_。
- 可以利用 \_\_\_\_\_ 方法修改线程的优先级。
- 线程可以通过调用 \_\_\_\_\_ 方法将处理器让给另一个优先级相同的线程。
- 如果要在等待一段时间之后再恢复执行一个线程, 应调用 \_\_\_\_\_ 方法。
- 通过调用 resume 方法, 可以使 \_\_\_\_\_ 线程重新激活。
- \_\_\_\_\_ 方法用于使等待队列中的第一个线程进入就绪状态。

- 13.2 判断下列句子是否正确。如果不正确，请解释原因。
- a) 如果一个线程停止了，那么它是不可运行的。
  - b) 在 Java 中，一个具有较高优先级的可运行线程将抢占优先级较低的线程。
  - c) Windows 和 Windows NT 上的 Java 系统采用时间分片方式，因此一个线程能够抢占优先级相同的其他线程。
  - d) 一个线程可以将处理器让给 (yield 方法) 优先级较低的线程。

## 自测练习答案

- 13.1 a) 单、多 b) 无用单元回收处理 c) 内存泄漏 d) stop e) 等待、睡眠、挂起、I/O 阻塞、f) 它的 run 方法运行结束，在该线程上调用 stop 方法 g) setPriority h) yield i) sleep j) 挂起 k) notify
- 13.2 a) 正确。  
b) 正确。  
c) 不正确。时间分片方式只允许一个线程执行一个时间分片，当超过该时间分片后，其他优先级相同的线程就可以执行了。  
d) 不正确。一个线程只能将处理器让给优先级相同的线程。

## 练习

- 13.3 判断下列句子是否正确。如果不正确，请解释原因。
- a) 当一个线程睡眠时，sleep 方法不消耗处理器时间。
  - b) 将一个方法声明为 synchronized 可以确保不会发生死锁。
  - c) Java 提供了一种功能强大的特性，称为多继承。
- 13.4 给出下列名词的定义。
- a) 线程
  - b) 多线程
  - c) 就绪状态
  - d) 阻塞状态
  - e) 抢占调度
  - f) Runnable 接口
  - g) 监控器
  - h) notify 方法
  - i) 生产者/消费者关系
- 13.5 a) 给出本章介绍的使用多线程的几个理由。  
b) 给出使用多线程的其他理由。
- 13.6 给出进入阻塞状态的 4 个原因。针对每个原因分别描述一下程序一般将如何退出阻塞状态，并进入可运行状态。
- 13.7 抢占调度和非抢占调度的区别是什么？Java 使用哪种调度策略？
- 13.8 什么是时间分片方式？给出支持时间分片方式的 Java 系统与不支持时间分片方式的 Java

系统在调度方式上的基本差别

- 13.9 一个线程为什么会调用 `yield` 方法?
- 13.10 开发 WWW 的 Java applet 时, 是什么原因促进了 applet 的设计者大量使用 `yield` 和 `sleep` 方法?
- 13.11 如果读者打算编写自己的 `start` 方法, 那么要想确保线程正确地启动, 应做些什么?
- 13.12 下面几种终止线程的方法有什么区别?
  - a) 忙等待
  - b) 睡眠
  - c) 挂起
- 13.13 编写一个 Java 程序, 判断一个线程是否为激活的
- 13.14
  - a) 什么是多继承?
  - b) 解释一个 Java 为什么不提供多继承?
  - c) Java 提供了什么功能来代替多继承?
  - d) 解释一下这种功能的典型用法
  - e) 这种功能怎样与抽象类进行区分?
- 13.15 `extends` 和 `implements` 的写法有什么区别?
- 13.16 在监控器的上下文中讨论下面各项的含义:
  - a) 监控器
  - b) 生产者
  - c) 消费者
  - d) `wait`
  - e) `notify`
  - f) `InterruptedException`
  - g) `synchronized`
- 13.17 ( 龟兔赛跑 ) 在第 5 章的练习中, 我们曾做过模仿龟兔赛跑的练习。现在让我们编写一个龟兔赛跑的新版本, 把乌龟和兔子分别放在单独的线程中。比赛开始时, 每个线程都调用 `start` 方法。使用 `wait`、`notify` 及 `notifyAll` 方法来同步它们的活动。
- 13.18 ( 多线程、网络、协作应用程序 ) 在第 16 章中, 我们将讨论 Java 的网络。一个多线程的 Java 应用程序可以并发地与几台主机通信。这样就有可能创建很多种有趣的协作应用程序。在学习第 16 章的网络知识之前, 先考虑几个可能采用多线程和网络技术的应用程序, 在学习了第 16 章之后, 再具体实现这些应用程序。
- 13.19 编写一个 Java 程序, 说明当一个高优先级的线程执行时, 将推迟所有低优先级线程的执行。
- 13.20 如果系统支持时间分片方式, 那么就编写一个 Java 程序, 说明在几个优先级相同的线程之间如何进行时间分片。该程序还应表示出一个低优先级线程的执行将由高优先级线程的时间分片推迟。
- 13.21 编写一个 Java 程序, 说明一个使用了 `sleep` 方法的高优先级线程可以使低优先级线程运行。
- 13.22 如果系统不支持时间分片方式, 那么就编写一个 Java 程序, 说明两个使用 `yield` 的线程可以使对方开始执行。
- 13.23 在像 Java 这样的允许线程等待的系统中, 很可能发生两个问题: 一个是死锁, 即一个或

多个线程永远等待某个不可能发生的事件;另一个是无限延迟,即延迟一个或多个线程的时间长度是不可预测的。编写一个例子,查看在一个多线程的Java程序中是怎样发生这些问题的。

- 13.24 (读者和作者)这个练习是开发一个Java监控器,去解决并发控制中的一个著名问题。这个问题是由P.J.Courtois、F. Heymans 和 D.L.Parnas 在他们的论文“Concurrent Control with Reader and Writers”中首次讨论并解决的(参见Communications of the ACM, Vol. 14, No. 10, October 1971, pp. 667-668),有兴趣的同学还可以阅读一下C.A.R.Hoare撰写的有关监控器的研究论文:“Monitors: An Operating System Structuring Concept”(参见Communications of ACM, Vol. 17, No. 10, October 1974, pp. 549-557; Corrigendum, Communications of the ACM, vol.18, No. 2, February 1975, p. 95。) [另外,在“Deitel, H.M., Operating Systems, Reading, MA: AddisonWesley, 1990”一书的第5章也对读者与作者的问题进行了详细讨论]

借助多线程,很多线程可以同时访问共享数据。我们已经看到,对共享数据的访问需要仔细地进行同步,以避免破坏数据。

假设有一个航空订票系统,顾客可以利用该系统订购两个城市之间的某次班机的机票。有关班机和座位的所有信息都保存在内存的一个公共数据库中。该数据库包括很多入口,分别表示在某一天两个城市之间的某一班飞机的某个座位。在典型的航空订票系统方案中,顾客将在数据库中不断查询,以查找满足顾客需要的“最佳”班机。因此,顾客可以在数据库中查询很多次,直到找到并订到一次班机。如果顾客找到了一个合适的座位但没有及时订票,那么这个座位很容易被其他人订走。这时,当顾客再想回来订票时,就会发现数据已经改变了,原来合适的那班飞机已被别人订走了。

查询数据的顾客称为读者,订购机票的顾客称为作者。显然,可以有任意多的读者同时查询共享数据,但每个作者对共享数据的访问必须是互斥的,以防止数据受到破坏。编写一个多线程的Java程序,激活多个读者线程和多个作者线程,每个线程都访问一个订票记录。一个作者线程有两个可能的事务:makeReservation 和 cancelReservation。一个读者有一个可能的事务:queryReservation。

首先实现一个允许对订票记录进行非同步访问的程序,看看会怎样破坏数据库的完整性。然后再使用Java监控器编写一个程序,使用wait和notify方法使访问共享订票记录的读者和作者按规定的协议达到同步。特别是当程序中没有活动的作者时,应允许多个读者同时访问共享数据。但如果有一个作者处于活动状态,那么就不允许任何读者访问共享数据。

注意,这个问题有很多微妙之处。例如,当存在着几个活动的读者,同时有一个作者想写入数据时,将会发生什么情况?如果我们一直允许固定流量的读者到达并共享数据,那么他们会无限地延迟作者的操作(作者可能会厌倦等待,从而将业务转移到其他地方)。为了解决这个问题,可以让读者等待作者。但这也不是真正的解决办法,因为固定流量的作者也会无限地延迟读者,使读者因为等待时间太久而将业务转移到其他地方。不妨使用下面几个方法来实现读者的监控器:一个是startReading方法,当读者想要开始访问订票数据时可以调用此方法;一个是stopReading方法,读者完成读取操作后可以调用此方法;一个是startWriting方法,当作者想要进行订票时可以调用此方法;一个是stopWriting方法;当作者完成订票时可以调用此方法。

- 13.25 编写一个程序，让一个篮球在 applet 中弹跳。使用一个 `mouseDown` 事件初始化该球，当篮球撞击 applet 的边缘时，应该以一个随机选择的角度（在 20 度到 60 度之间）反弹回去。
- 13.26 修改练习 13.25，允许用户在篮球弹跳时在 applet 上画出 5 条线，当篮球撞上某条线时，应该以一个随机选择的角度（在 20 度到 60 度之间）从线上反弹回去。

## 第14章 多媒体：图像、动画和声音

### 教学目标

- 理解如何获取和显示图像
- 学会根据一系列图像来创建动画，学会控制动画的速度和闪烁
- 学会获取、播放、循环和停止声音
- 学会使用 MediaTracker 来监视图像的加载，学会创建图像映射
- 学会使用 param 标记定制 applet

### 14.1 简介

欢迎进入多媒体的世界！多媒体可能成为计算机历史上最大的一次革命。几十年前，我们使用计算机的目的，主要是对它能够进行高速算术计算感兴趣。但是随着计算机领域的不断发展，我们开始意识到：计算机的数据操作能力与其计算能力同等重要。Java的“热点”是多媒体，它利用声音、图像、图形和视频使应用程序变得“生动活泼”。目前，很多人都把二维彩色视频看成是多媒体的“极限”。但在近10年中，我们一直在尝试着开发各种令人激动的三维应用程序。

多媒体编程向人们提出了很多新的挑战，这个领域已经拓展得非常庞大，而且还将迅速地扩展下去。人们都争着为自己的计算机添加多媒体设备。如今，大多数待销售的新计算机都已经带有多媒体配置，即带有CD或DVD驱动器、声卡或者其他视频功能。

对于那些需要图形的用户来说，二维的图形已无法再满足要求。现在，很多人都要求图形是三维、高分辨率和彩色的。在今后的10年中，开发出真正的三维图像将成为可能。我们可以想像一下，如果拥有了超高清晰度、“环绕剧场”式的三维电视，那么转播的各种体育和娱乐活动就像发生在我们身边一样！世界各地的医科学生可以同时看到远在千里之外进行的手术，仿佛他们都在同一房间。人们可以先在房间中利用逼真的驾驶模拟器来学习驾驶，然后再驾驶真正的汽车。类似的这种令人激动的情景将数不胜数！

实现多媒体需要非凡的计算能力。不久之前，具备这种计算能力且人们能够负担的计算机还没有出现。而现在，各种超速处理器的出现使高效的多媒体技术得以实现，这些处理器包括Sun Microsystems公司的SPARC Ultra芯片、Intel公司的Pentium芯片和Itanium芯片、DEC公司的Alpha芯片以及MIPS/Silicon Graphics公司的处理器等。计算机厂商和通信设备厂商将成为多媒体革命的受益者。用户们也一定非常乐意为更快的处理器、更大的内存和更宽的通信带宽而付出更多的钱，因为这些都是运行多媒体应用程序所必需的。不过，事实上用户可能根本不必付出更多的钱，因为这些厂商之间的激烈竞争会导致价格下降。

我们还需要能够轻易地实现多媒体技术的编程语言。大多数编程语言都没有内置的多媒体功能，但是，Java通过一些类软件包（它们是Java编程世界的一个组成部分）提供了扩展的多媒体功能。利用这些功能，我们可以立即开始进行功能强大的多媒体应用程序的开发！

本章提供了一系列生动的小例子，它们包含了很多有趣的多媒体特性，可用于开发有用的应用





```
16
17     // display the image
18     public void paint( Graphics g )
19     {
20         // draw the original image
21         g.drawImage( deitel, 1, 1, this );
22
23         // draw the image with its width and height doubled
24         int width = deitel.getWidth( this );
25         int height = deitel.getHeight( this );
26         g.drawImage( deitel, 1, 90, width * 2,
27                     height * 2, this );
28     }
29 }
```



图 14.1 在 applet 中加载和显示一个图像

在 applet 的 paint 方法中，使用 Graphics 的 drawImage 方法来显示图像。在这个 applet 中，使用了两个 drawImage 语句。下面是一个带有四个参数的 drawImage 方法：

```
g.drawImage ( deitel , 1, 1, this ) ;
```

第一个参数表示要显示的图像对象；第二个参数和第三个参数表示该图像在 applet 上的显示位置，分别为 x 坐标和 y 坐标（这个坐标表示的是图像左上角的显示坐标）；最后一个参数是指向一个 ImageObserver 对象的引用。通常情况下，ImageObserver 是指将要显示图像的对象，在本例中，this 是指当前的 applet。ImageObserver 是 Component 类（Applet 类的一个间接超类）提供的一个接口，它可以是任何一个提供了 ImageObserver 接口的对象。在显示需要从 Internet 上下载很长时间的大图像时，这个参数是很重要的。一个程序很可能在图像还没有完全下载时就将其显示出来，当图像的其余部分都下载完毕时，系统会自动通知 ImageObserver 更新已显示的图像。运行本例时，要仔细地观察该图像是边加载、边显示的。

下面的语句是使用 drawImage 方法将该图像按比例放大后显示出来：

```
g.drawImage ( deitel, 1, 90, width * 2, height * 2 , this );
```

第四个参数和第五个参数分别表示图像的显示宽度和高度。Java 将自动调整图像的大小，使其适合

指定的宽度和高度，这里我们将该图像的宽度和高度分别放大一倍。该图像的宽度和高度是由 Image 的 getWidth 方法和 getHeight 方法确定的，这两个方法都带有一个 ImageObserver 参数。这样，随着 Image 加载到 applet 中，图像可以不断进行更新。

### 14.3 动画介绍：图像的循环

图 14.2 中的 applet 说明了一个简单的动画，该 applet 使用与图 14.1 中的程序相同的方法来加载和显示图像。在后面的几个例子中，为了使动画保持流畅，还使用了几种其他的方法。

```

1 // Fig. 14.2: DeitelLoop.java
2 // Load an array of images, loop through the array,
3 // and display each image.
4 import java.applet.Applet;
5 import java.awt.*;
6
7 public class DeitelLoop extends Applet {
8     private Image deitel[];
9     private int totalImages = 30, // total number of images
10         currentImage = 0, // current image subscript
11         sleepTime = 40; // milliseconds to sleep
12
13     // load the images when the applet begins executing
14     public void init()
15     {
16         deitel = new Image[ totalImages ];
17
18         for ( int i = 0; i < deitel.length; i++ )
19             deitel[ i ] = getImage( getDocumentBase(),
20                 "images/deitel" + i + ".gif" );
21     }
22
23     // start the applet
24     public void start()
25     {
26         currentImage = 0; // always start with 1st image
27     }
28
29     // display the image in the Applet's Graphics context
30     // then sleep and call repaint
31     public void paint( Graphics g ,
32     {
33         g.drawImage( deitel[ currentImage ], 1, 1, this );
34
35         // fix to help load images in Netscape Navigator
36         // makes browser "think" there is a mouse event
37         postEvent( new Event( this, Event.MOUSE_ENTER, " ) );
38
39         currentImage = ++currentImage % totalImages;
40
41         try {
42             Thread.sleep( sleepTime );
43         }

```

```

44         catch ( InterruptedException e ) {
45             showStatus( e.toString() );
46         }
47
48         repaint();
49     }
50 }

```

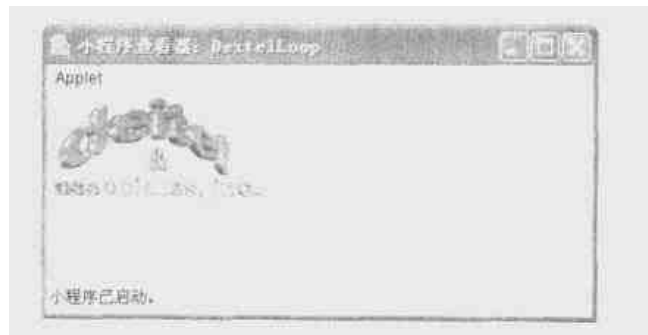


图 14.2 一个使用图像循环的简单动画

DeitelLoop 类中包括一个 Image 数组，这些图像都是在 applet 的 init 方法中加载的。下面的语句是使用 getImage 方法来加载每个独立的图像（该动画共有 30 个独立的图像）：

```
deitel[i] = getImage( getDocumentBase( ) , " images/deitel " + i + " .gif " );
```

第二个参数是用字符串“image/deitel”、i 和“.gif”来组合文件名，动画中的每个图像都是文件“deitel0.gif”到文件“deitel29.gif”之中的一个，利用 for 循环中的控制变量来选择。记住，每加载一幅图像，getImage 方法都会启动一个独立的线程。

#### 性能提示 14.1

将动画的多幅图像（多个帧）组合成一个图像一次加载，要比分别加载每一个图像高效得多（可以利用一个画图程序将动画的多个帧组合到一个图像中）。每调用一次 getImage 方法，都需要对保存图像的 WWW 站点进行一次单独的连接。

#### 性能提示 14.2

如果将动画的所有帧作为一个大图像一次加载，那么程序就不得不在动画显示之前进行等待。

本例中，显示动画的工作是由 applet 的 paint 方法完成的，实例变量 currentImage 用于保存当前所显示的图像。首先，使用 drawImage 方法显示一幅图像，然后利用下面的语句使 applet 睡眠片刻：

```
Thread.sleep( sleepTime );
```

接着调用 repaint 方法，repaint 方法再调用该 applet 的 update 方法，清除该 applet 的背景。然后，update 方法再调用 paint 方法显示下一幅图像。

下面的语句有助于动画图像在流行的浏览器中正确地加载：

```
postEvent ( new Event( this , Event . MOUSE_ENTER , " " ) );
```

当在浏览器中加载了很多图像时，applet 通常会暂停，不再加载任何内容。这时，如果使用鼠标在屏幕上的 applet 中移进再移出，就可以加载图像了，继而动画开始执行。如同第 10 章所介绍的，当鼠标指针移进和移出 applet 时，系统会产生 MOUSE\_ENTER 和 MOUSE\_EXIT 事件。上面的语句是使用 postEvent 方法（从 Component 类间接继承的）使 applet “认为”发生了一个 MOUSE\_ENTER 事件。这样，图像就能够在浏览器中正确地加载。执行 Java appletviewer 中的 applet 时不需要这个

语句。在该语句中，创建新的 Event 对象的构造函数带有三个参数：一个是发生该事件的组件（即这个 applet），一个是事件 ID（Event.MOUSE\_ENTER），一个是保存有关该事件信息的 Object 引用（本例中使用的是空字符串）。

执行该 applet 时，我们会注意到，图像的加载需要花费一些时间，这通常会导致只显示一部分图像。在显示每个图像时仔细地观察，可能发现每个图像都是一片一片显示的（这与图 14.1 的例子类似）。这是由所使用的图像格式引起的。例如，GIF 图像可以利用交错格式和非交错格式进行保存。这两种格式都是指定图像像素的保存顺序。一个非交错格式的图像是将像素完全按它们出现在屏幕上的顺序保存。当显示一个非交错格式的图像时，它会随着像素信息的读取而从上到下一块一块地出现。一个交错格式的图像是将像素按行保存，但各行之间是无序的。例如，图像中的各像素行可以按照第 1 行、第 5 行、第 9 行、第 13 行……，然后第 2 行、第 6 行、第 10 行、第 14 行……的顺序保存。当显示这种格式的图像时，首先会显示第一批像素行，使图像的大致轮廓出现；然后再显示第二批像素行，将图像进行细化；依次类推，从而使整个图像逐渐清晰，直到全部显示出来。

另外还要注意，在显示每个图像时，动画都会闪烁。这是由于调用了 applet 的 update 方法。update 清除 applet 的方法是使用当前的背景色绘制一个占满 applet 的填充矩形，这样可以把刚画的图像覆盖。这样，整个执行过程就是：applet 绘制一个图像，睡眠片刻，清除背景（产生闪烁），然后再绘制下一幅图像。

在这个 applet 中，似乎出现了一个无限循环——paint 调用 repaint，repaint 调用 update，update 调用 paint，而 paint 又调用 repaint；实际上并非如此。每次调用 repaint 都将启动一个独立的线程（它会创建一个独立的方法调用堆栈）来完成 applet 的重画。这个“无限循环”实际是这样执行的：repaint 启动一个新线程，调用 update，update 调用 paint，paint 调用 repaint，repaint 再启动一个新线程调用 update，上一个重画线程终止。每个线程的方法调用堆栈中永远不会超过三个方法。而且，当一个线程进行无用单元回收处理时，它的调用堆栈也会自动进行无用单元回收处理。尽管这种“无限循环”是表面上的，但还是有问题：如果用户移到另一个网页上，那么这个 applet 仍将继续使用计算机的处理器时间，这可能会降低计算机的性能。

#### 性能提示 14.3

如果 applet 是通过在 paint 方法中调用 repaint 来实现动画，那么即使用户离开了该 applet 所在的网页，该 applet 也会继续占用处理器时间。这会导致用户计算机的性能降低。

#### 软件工程观点 14.1

创建动画时一定要提供一个取消动画的机制。这样，当用户离开动画 applet 所在的网页而进入另一个网页时，就可以利用该机制而使原来的动画停止。在后面的例子中，我们将讨论几种方法，以弥补前面提到的有关动画的一些不足。

## 14.4 图形双缓存

开发基于多媒体的应用程序时，用户一定很关心声音和动画是否流畅。不连贯的声像效果很难让人接受，如果是直接向屏幕上绘图，通常会发生这种不连贯的情况。采用图形双缓存技术可以避免这种情况，即当程序向屏幕上显示一幅图像时，它可以同时在屏幕外的一个缓冲区中创建下一幅图像。这样，当显示下一幅图像时，就可以流畅地把它放到屏幕上了。当然，这是一种以空间换时间的方法，需要大量的额外内存空间，但它能够大大地提高显示性能，因而还是很值得的。

在使用其他方法进行绘图的程序中(我们前面介绍的例子都是在paint方法中进行绘图),图形双缓存也是很有用的。屏幕外缓冲区也可以在方法之间,甚至在不同的类对象之间传递,以便使其他的方法或对象能够在该屏幕外的缓冲区上进行绘图,随后就可以显示出绘图的结果。

#### 性能提示 14.4

双缓存可以减少或消除动画的闪烁,但它会明显地降低动画的运行速度。

当一个图像的所有像素没有一次显示时,动画会产生更多的闪烁。如果利用图形双缓存来绘制一个图像,那么到显示该图像时,已经在屏幕外绘制完该图像。这样,对于用户来说,图像的渐显显示或块状显示效果将被隐藏起来。在用户眼中,所有的像素都是同时显示的,这样,闪烁就可以减到最小或完全将其消除。

图形双缓存的基本概念是这样的:创建一个空的Image,在这个空的Image上绘图(使用Graphics类的方法),然后显示该图像。

图 14.3 中的 applet 对图 14.2 的动画进行了扩展,增加了图形双缓存功能。实现一个图形双缓存有两个关键点是 Image 引用(本例中为 buffer)和 Graphics 引用(本例中是 gContext)。Image 用于保存要显示的实际像素,Graphics 引用用于绘制像素。每个图像都有一个相关的图形环境——即一个 Graphics 类对象,用于完成绘图。图形双缓存所用的 Image 和 Graphics 引用通常称为屏外图像和屏外图形环境,因为它们实际上操作的不是屏幕上的像素。

```
1 // Fig. 14.3: DeitelLoop2.java
2 // Load an array of images, loop through the array,
3 // and display each image.
4 import java.applet.Applet;
5 import java.awt.*;
6
7 public class DeitelLoop2 extends Applet {
8     private Image deitel [ ];
9     private int totalImages = 30, // total number of images
10             currentImage = 0, // current image subscript
11             sleepTime = 40; // milliseconds to sleep
12
13     // The next two objects are for double-buffering
14     private Graphics gContext; // off-screen graphics context
15     private Image buffer; // buffer in which to draw image
16
17     // load the images when the applet begins executing
18     public void init()
19     {
20         deitel = new Image [ totalImages ];
21         buffer = createImage( 160, 80 ); // create image buffer
22         gContext = buffer.getGraphics(); // get graphics context
23
24         // set background of buffer to white
25         gContext.setColor( Color.white );
26         gContext.fillRect( 0, 0, 160, 80 );
27
28         for ( int i = 0; i < deitel.length; i++ )
29             deitel[ i ] = getImage( getDocumentBase(),
30                                     "images/deitel" + i + ".gif" );
31     }
32 }
```

```

33      // start the applet
34      public void start()
35      {
36          // always start with 1st image
37          gContext.drawImage( deitel [ 0 ], 0, 0, this );
38          currentImage = 1;
39      }
40
41      // display the image in the Applet's Graphics context
42      public void paint( Graphics g )
43      {
44          g.drawImage( buffer, 0, 0, this );
45
46          // clear previous image from buffer
47          gContext.fillRect( 0, 0, 160, 80 );
48
49          // draw new image in buffer
50          gContext.drawImage( deitel [ currentImage ], 0, 0, this );
51
52          // fix to help load images in Netscape Navigator
53          // makes browser "think" there is a mouse event
54          postEvent( new Event( this, Event.MOUSE_ENTER, "" ) );
55
56          currentImage = ++currentImage % totalImages;
57
58          try {
59              Thread.sleep( sleepTime );
60          }
61          catch ( InterruptedException e ) {
62              showStatus( e.toString() );
63          }
64
65          repaint(); // display buffered image
66      }
67  }

```

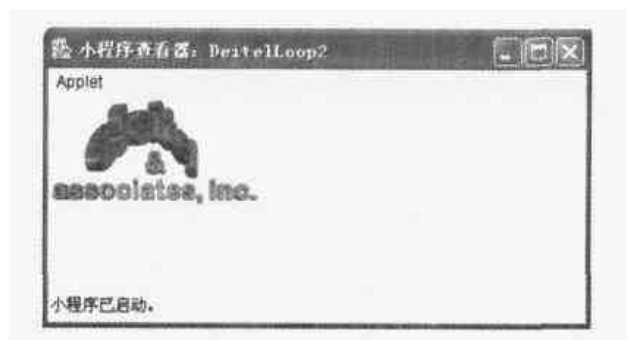


图 14.3 使用图形双缓存功能来减少动画闪烁

在 applet 的 init 方法中，利用下面的语句来创建图形双缓存：

```

buffer = createImage ( 160,80 );// create image buffer
gContext = buffer.getGraphics ( );// get graphics context

```

第一条语句是使用该 applet 的 createImage 方法（从 Component 类间接继承）创建一个宽 160 像素、高 80 像素的图像（这是在动画中将要显示的每个图像的大小）。Image 类是一个抽象类，所以不能

直接创建 Image 对象。为了实现运行平台的独立性, Java 在每个平台的版本上都提供了自己的 Image 子类, 用于保存图像信息。Image 类为程序员提供了一套处理图像的统一方法。createImage 方法用于创建该平台的 Image 子类的对象。第二条语句使用 Image 类的 getGraphics 方法, 获取与 buffer 引用的 Image 对象相关的图形环境 (即 Graphics 对象)。gContext 引用将用于绘制屏外图像。下面两条语句将图形环境的绘图颜色设置成白色 (任何图形都利用该颜色进行绘制), 然后向屏外图像上绘制一个白色的填充矩形:

```
gContext.setColor ( Color . white );
gContext.fillRect (0, 0, 160, 80 );
```

这样, 这个屏外图像就做好了绘制第一幅动画图像的准备。

start 方法指定图像数组中的第一个图像应该使用 gContext 进行绘制, 该图像保存在由 buffer 引用的 Image 对象中。paint 方法先将保存在 buffer 中的当前图像显示出来, 然后再准备下一幅图像。首先, 使用 fillRect 方法清除屏外图像。然后, 将下一幅图像绘制到屏外图像上。睡眠片刻后, 再调用 repaint 方法。这将启动一个调用 update 方法的独立线程, update 再调用 paint。在 paint 方法中, 首先显示已准备好的图像, 然后再准备下一个要显示的图像。

当执行这个 applet 时, 图像加载时动画仍然是不连贯的。不过, 一旦将图像加载完, 在前面的动画 applet 中出现的闪烁大部分都会消除掉。在这个 applet 中, 仍然在 paint 中调用了 repaint 方法。这意味着当用户转移到另一个网页上时, 该 applet 仍将占用用户计算机的处理器时间。在下一个例子中, 我们将消除所有的闪烁。

## 14.5 利用 MediaTracker 来监视图像的加载

在前面的两个动画中我们已经看到, 当加载图像时, 经常会只显示出图像的一部分, 这就使动画在开始时出现不连贯。在图 14.4 的 applet 中, 我们扩展了该动画, 使用 Java 的 MediaTracker 来判断一个图像何时完成加载。一旦图像完成了加载, 就可以将其作为动画的一帧显示出来。

```
1 // Fig. 14.4: DeitelLoop3.java
2 // Load an array of images, loop through the array,
3 // and display each image.
4 import java.applet.Applet;
5 import java.awt.*;
6
7 public class DeitelLoop3 extends Applet {
8     private Image deitel [ ];
9     private int totalImages = 30, // total number of images
10         currentImage = 0, // current image subscript
11         sleepTime = 40; // milliseconds to sleep
12
13     // The next two objects are for double-buffering
14     private Graphics gContext; // off-screen graphics context
15     private Image buffer; // buffer in which to draw image
16
17     private MediaTracker imageTracker; // used to track images
18
19     // load the images when the applet begins executing
20     public void init()
21     {
```



```

22         dImage = new Image( totalImages );
23         buffer = createImage( 160, 80 ); // create image buffer
24         gContext = buffer.getGraphics(); // get graphics context
25
26         // set background of buffer to white
27         gContext.setColor( Color.white );
28         gContext.fillRect( 0, 0, 160, 80 );
29
30         imageTracker = new MediaTracker( this );
31
32         for ( int i = 0; i < deitel.length; i++ ) {
33             deitel[i] = getFrame( getDocumentBase(),
34                 "images/deitel " + i + ".gif" );
35
36             // track loading image
37             imageTracker.addImage( deitel[i], i );
38
39         }
40         try {
41             imageTracker.waitForID( 0 );
42         }
43         catch ( InterruptedException e ) { }
44     }
45
46     // start the applet
47     public void start()
48     {
49         // always start with 1st image
50         gContext.drawImage( deitel[0], 0, 0, this );
51         currentImage = 1;
52     }
53
54     // display the image in the Applet's Graphics context
55     public void paint( Graphics g )
56     {
57         g.drawImage( buffer, 0, 0, this );
58
59         if ( imageTracker.checkID( currentImage, true ) ) {
60             // clear previous image from buffer
61             gContext.fillRect( 0, 0, 160, 80 );
62
63             // draw new image in buffer
64             gContext.drawImage(
65                 deitel[ currentImage ], 0, 0, this );
66
67             currentImage = ++currentImage % totalImages;
68         }
69         else // fix to help load images in Netscape Navigator
70             postEvent( new Event( this, Event.MOUSE_ENTER, "" ) );
71
72         try {
73             Thread.sleep( sleepTime );
74         }
75         catch ( InterruptedException e ) {
76             showStatus( e.toString() );
77         }

```

```
78
79     repaint(); // display buffered image
80 }
81
82 // override update to eliminate flicker
83 public void update( Graphics g )
84 {
85     paint( g );
86 }
87 }
```



图 14.4 利用 MediaTracker 使动画流畅地启动，并重写 update 方法，消除动画闪烁

#### 性能提示 14.5

一些使用过 MediaTracker 对象的用户曾经报告，MediaTracker 对象会对性能带来不利影响。请记住这一点，以便在必要时调整自己的多媒体应用程序。

在向一个程序中加载图像时，这些图像可以使用 MediaTracker 类的一个对象进行登记，以使程序能够确定一个图像何时加载完。MediaTracker 还提供了下面的功能：在程序继续执行之前等待一个或几个图像加载，以及确定加载图像时是否产生了错误。

MediaTracker 对象通过下面的语句来创建：

```
imageTracker = new MediaTracker (this );
```

该构造函数的参数是一个 ImageObserver（各图像将绘制在该对象上）。在这里，applet（this）是 ImageObserver。使用 getImage 方法加载了每个图像之后，就执行下面的语句，以使用 MediaTracker 的对象 imageTracker 来登记已加载的对象：

```
imageTracker.addImage ( deitel [ i ] , i );
```

addImage 方法的第一个参数是已加载对象的 Image 引用。第二个参数是一个整数 ID，可用于标识该图像，该 ID 不必是惟一的。如果要想将多个图像作为一个组来处理，只需为它们设置同样的 ID 值。

一旦所有的图像都已经利用 imageTracker 登记了，那么执行下面的语句就可以强制程序进行等待，直到标识为 0 的图像加载完成：

```
imageTracker.waitForID(0);
```

这是一个阻塞操作，因为在图像加载完成之前，程序无法继续执行。MediaTracker 类还提供了一个 waitAll 方法，它将阻塞程序的执行，直到所有登记的图像都加载完成。

#### 性能提示 14.6

如果使用 MediaTracker 的 waitAll 方法等待所有登记的图像完成加载，那么从 applet 开始执行到图像实

际显示出来会有很长的一段延迟。图像越多、越大，用户需要等待的时间就越长。因此，MediaTracker 的 waitAll 方法只能用于等待数目较少的图像完成加载。

#### 编程技巧 14.1

在加载图像的同时，屏幕上应显示一些其他内容，不要让屏幕空白。因为用户在看到屏幕上的信息之前等待的时间越长，他们就越有可能在信息出现之前离开该网页。

第一页加载完成后，就在 start 方法中准备显示，然后再在 paint 方法中实际显示。下一幅图像的准备工作在 paint 方法的 if 结构中完成。下面给出了该 if 结构的判断条件：

```
imageTracker.checkID ( currentImage , true )
```

这个判断条件完成两个任务：首先，使用 checkID 方法检查登记为 currentImage 的图像，确定该图像是否已加载完成。如果已加载完成，该方法就返回 true，并执行 if 结构体，准备下一幅要显示的图像。如果该图像还没有完全加载，该方法就返回 false，执行 if/else 结构的 else 部分。如果在调用 checkID 检查图像时，该图像还没有开始加载，那么 checkID 的第二个参数就表示该图像现在应该开始加载。

MediaTracker 并没有消除闪烁——它只是使动画能够在加载图像的同时流畅地启动，这样图像就不会只显示一部分。为了消除所有的闪烁，我们提供了一个自定义的 update 方法。以前，一般使用 update 方法来清除 applet，然后再调用 paint 方法。清除 applet 会导致闪烁。我们重写了从 Component 类间接继承的 update 方法。在新的 update 方法中，只是简单地调用 paint——没有对 applet 进行清除。那么，怎样清除 applet 上的图像，以便显示下一幅图像呢？其实，我们已经完成了这个处理过程。每当我们准备下一幅图像时，一定会先清除屏外图像，使它的背景都是白色，然后再在白色背景上绘制下一幅图像。这样，当显示该屏外图像时，它的白色背景就会将 applet 上已有的图像完全覆盖。

当执行着 applet 时，闪烁已经完全消除了！不过，这个动画的第一遍演示（即第一次显示每幅图像时）要比随后的演示慢。这是因为我们使用了 MediaTracker 来等待每个图像的加载。在下面的例子中，我们增加了一个独立的线程来执行动画，这样就可以控制动画何时执行和何时停止了。

## 14.6 利用一个独立线程来运行动画

图 14.5 的 applet 没有增添任何新的多媒体或动画技术，但它展示了一种更好的创建动画的方法，使得可以“友好地浏览”动画。我们前面已经提到，即使用户离开了当前网页，动画仍然会继续执行，并占用用户计算机的处理器时间。在这个例子中，我们对 applet 进行了扩展，提供了 Runnable 接口，这样就可以在一个独立的线程上运行动画，并且可以控制该线程的执行。

```
1 // Fig. 14.5: DeitelLoop4.java
2 // Load an array of images, loop through the array,
3 // and display each image.
4 import java.applet.Applet;
5 import java.awt.*;
6
7 public class DeitelLoop4 extends Applet implements Runnable {
8     private Image deitel[];
9     private int totalImages = 30,    // total number of images
```

```

10         currentImage = 0;        // current image subscript
11         sleepTime = 40;           // milliseconds to sleep
12
13         // The next two objects are for double-buffering
14         private Graphics gContext; // off-screen graphics context
15         private Image buffer;      // buffer in which to draw image
16
17         private MediaTracker imageTracker; // used to track images
18
19         private Thread animate;      // animation thread
20         private boolean suspended;   // toggle on/off
21
22         // load the images when the applet begins executing
23         public void init()
24         {
25             deitel = new Image[ totalImages ];
26             buffer = createImage( 160, 80 ); // create image buffer
27             gContext = buffer.getGraphics(); // get graphics context
28
29             // set background of buffer to white
30             gContext.setColor( Color.white );
31             gContext.fillRect( 0, 0, 160, 80 );
32
33             imageTracker = new MediaTracker( this );
34
35             for ( int i = 0; i < deitel.length; i++ ) {
36                 deitel[ i ] = getImage( getDocumentBase(),
37                     "images/deitel" + i + ".gif" );
38
39                 // track loading image
40                 imageTracker.addImage( deitel[ i ], i );
41             }
42
43             try {
44                 imageTracker.waitForID( 0 );
45             }
46             catch( InterruptedException e ) { }
47         }
48
49         // start the applet
50         public void start()
51         {
52             // always start with 1st image
53             gContext.drawImage( deitel[ 0 ], 0, 0, this );
54             currentImage = 1;
55
56             // create a new animation thread when user visits page
57             if ( animate == null ) {
58                 animate = new Thread( this );
59                 animate.start();
60             }
61         }
62
63         // terminate animation thread when user leaves page

```

```
64     public void stop()
65     {
66         if ( animate != null ) {
67             animate.stop();
68             animate = null;
69         }
70     }
71
72     // display the image in the Applet's Graphics context
73     public void paint( Graphics g )
74     {
75         g.drawImage( buffer, 0, 0, this );
76     }
77
78     // override update to eliminate flicker
79     public void update( Graphics g )
80     {
81         paint( g );
82     }
83
84     public void run()
85     {
86         while ( true ) {
87             if ( imageTracker.checkID( currentImage, true ) ) {
88                 // clear previous image from buffer
89                 gContext.fillRect( 0, 0, 160, 80 );
90
91                 // draw new image in buffer
92                 gContext.drawImage(
93                     deitel[ currentImage ], 0, 0, this );
94
95                 currentImage = ++currentImage % totalImages;
96             }
97             else // browser fix: help load images
98                 postEvent(
99                     new Event( this, Event.MOUSE_ENTER, "" ) );
100
101             try {
102                 Thread.sleep( sleepTime );
103             }
104             catch ( InterruptedException e ) {
105                 showStatus( e.toString() );
106             }
107
108             repaint(); // display buffered image
109         }
110     }
111
112     public boolean mouseDown( Event e, int x, int y )
113     {
114         if ( suspended ) {
115             animate.resume();
116             suspended = false;
117         }
```

```
118         else {
119             animate.suspend();
120             suspended = true;
121         }
122
123         return true;
124     }
125 }
```

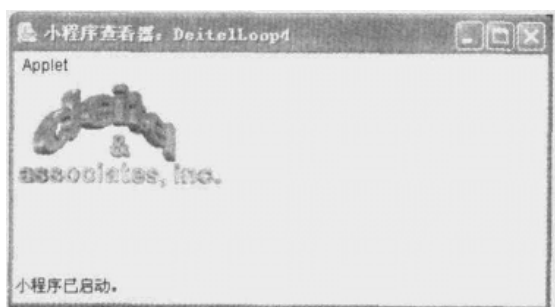


图 14.5 利用一个独立线程来运行动画

Thread 对象 `animate` 用于运行动画，布尔变量 `suspended` 用于标识 applet 当前是否处于暂停状态。当 applet 的 `start` 方法开始执行时，首先在屏外图形环境上绘制第一幅图像，然后创建一个 Thread（利用这个 applet 进行初始化），并调用该线程的 `start` 方法，这样就启动了该动画线程。applet 的 `run` 方法用于控制动画。原来由 `paint` 方法完成的功能现在都由 `run` 方法中的无限循环语句来完成，只有图像的实际显示仍然是由 `paint` 完成的。

还有另外两个关键的方法——`stop` 方法和 `mouseDown` 方法。记住，当用户离开 applet 所在的网页时，系统会自动调用该 applet 的 `stop` 方法。在 `stop` 方法中，调用了线程的 `stop` 方法来终止当前线程。这样，当用户移到另一个网页时，applet 就不会继续运行动画了。如果用户重新访问包含该 applet 的网页，则系统会自动调用 `start` 方法，因此 `start` 方法将创建一个新的线程来运行动画。如果发生了这种情况，那么动画会立刻开始运行，因为所有的图像仍然装载在内存中。通过使用 `mouseDown` 方法，用户可以使用鼠标点击 applet 来暂停动画，然后再点击一次就可以恢复动画，这是分别使用 Thread 的 `suspend` 方法和 `resume` 方法完成的。这两个功能的增加使动画 applet 的浏览方式更友好。注意，布尔变量 `suspended` 还没有进行初始化，因此我们使用 Java 的默认初始值 `false`。

## 14.7 加载和播放音频剪辑

Java 程序可以操作和播放音频剪辑。现在，用户可以很容易地获取自己的音频剪辑，而且从各种软件产品和 Internet 上也可以获得大量可用的音频剪辑。因此，系统必须配置音频硬件——典型的有扬声器和声卡。

Java 提供了两种在 applet 中播放音频的机制——Applet 类的 `play` 方法及 AudioClip 接口的 `play` 方法。如果想要在程序中播放某种音频，可以使用 Applet 类的 `play` 方法，将音频加载进来，然后播放；当播放完音频后，就对其设置无用单元回收处理标记。Applet 类的 `play` 方法有以下两种形式：

```
public void play ( URL location, String soundFileName );
public void play ( URL soundURL );
```

第一种形式是将保存在文件 `soundFileName` (来自 `location`) 中的音频剪辑加载进来, 然后播放这个音频。第一个参数通常是调用 `getDocumentBase` 方法或 `getCodeBase` 方法。`getDocumentBase` 方法用于指示将加载 applet 的 HTML 文件放在什么位置。`getCodeBase` 方法用于指示将 applet 的 class 文件放在何处。第二种形式的 `play` 方法只有一个 URL 参数, 它包含音频剪辑的文件名及所在位置。下面的语句是将文件 `hi.au` 中的音频剪辑加载进来并进行播放:

```
public void play (getDocumentBase ( ) , " hi.au" ) ;
```

图 14.6 的程序说明了如何加载和播放一个 `AudioClip`。这种方法更加灵活。它允许将声音保存在程序中, 这样, 当程序执行时就可以重新播放这段声音。`Applet` 类的方法 `getAudioClip` 有两种形式, 其参数与前面介绍的 `play` 方法一样, `getAudioClip` 方法返回一个 `AudioClip` 引用。将 `AudioClip` 加载进来后, 就可以调用该对象的 3 个方法——`play`、`loop` 和 `stop`。`play` 方法用于将音频播放一次。`loop` 方法用于在背景中不断地循环播放该音频剪辑。`stop` 方法用于终止当前播放的音频剪辑。在这个程序中, 每个方法都对应着 applet 上的一个按钮:

```
1 // Fig. 14.6: LoadAudioAndPlay.java
2 // Load an audio clip and play it.
3 import java.applet.*;
4 import java.awt.*;
5
6 public class LoadAudioAndPlay extends Applet {
7     private AudioClip sound;
8     private Button playSound, loopSound, stopSound;
9
10    // load the image when the applet begins executing
11    public void init()
12    {
13        sound = getAudioClip( getDocumentBase(), "hi.au" );
14        playSound = new Button( "Play" );
15        loopSound = new Button( "Loop" );
16        stopSound = new Button( "Stop" );
17        add( playSound );
18        add( loopSound );
19        add( stopSound );
20    }
21
22    public boolean action( Event e, Object o )
23    {
24        if ( e.target == playSound )
25            sound.play();
26        else if ( e.target == loopSound )
27            sound.loop();
28        else if ( e.target == stopSound )
29            sound.stop();
30
31        return true;
32    }
33 }
```



图 14.6 加载和播放一个音频剪辑

## 14.8 通过 HTML 的 param 标记来定制 applet

在浏览 WWW 时,我们经常会发现公共区中的一些 applet,可以将它们直接应用在自己的网页上以节省费用(一般情况下,这些 applet 的构造函数应该是可以信任的),这种 applet 的一种常用特性是,通过引用该 applet 的 HTML 文件所提供的参数来定制该 applet。例如,下面的 HTML 文件调用了 appletDeitelLoop5 (如图 14.7 所示),并指定了 3 个参数:

```
<html>
<applet code = "DeitelLoop5.class" width = 400 height = 400>
<param name = " totalimages " value = " 30 ">
<param name = " imagename " value = " deitel " >
<param name = " sleeptime " value = " 500 " >
</applet>
</html>
```

param 标记行必须出现在 applet 的开始标记和结束标记之间,这些值随后就将用于定制 applet。在 applet 的开始标记和结束标记之间,可以出现任意多个 param 标记,每个标记中都有一个 name 值和一个 value 值。applet 的 getParameter 方法用于获取与某个参数相关的 value 值,并将该 value 作为一个字符串返回。getParameter 的传入值是一个字符串,即该参数的名称。例如,下面的语句是获取与 sleeptime 参数相关的值,并将该值以字符串形式赋给 parameter;

```
parameter = getParameter ( " sleeptime " );
```

如果没有包含指定参数的 param 标记行,那么 getParameter 将返回 null。

在图 14.7 的动画 applet 中,允许用户使用他们自己的图像来定制动画。这个 applet 提供了 3 个参数。参数 sleeptime 是指显示两幅图像之间的间隔(睡眠)时间(单位为毫秒)。这个值将转换成一个整数,作为实例变量 sleepTime 的值。参数 imageName 是要加载图像的基本名称。这个字符串将赋给实例变量 imageName。这个 applet 假定各图像都在一个子目录 images 中,它与 applet 位于同一个目录中。这个 applet 还假定图像文件名是从 0 开始编号。参数 totalimages 表示动画中图像的总数。它的值将转换成一个整数,并赋给实例变量 totalImages。

这个 applet 还提供了文本字段,用于显示睡眠时间,并允许用户向文本字段中输入新值来修改睡眠时间。这是对前面使用独立线程来运行动画的 applet 的扩展。

```
1 // Fig. 14.7: DeitelLoop5.java
2 // Load an array of images, loop through the array,
```



```
3 // and display each image. This version is customizable.
4 //
5 // HTML parameter "sleepTime" is an integer indicating the
6 // number of milliseconds to sleep between images.
7 //
8 // HTML parameter "imageName" is the base name of the images
9 // that will be displayed (i.e., "deitel" is the base name
10 // for images "deitel0.gif," "deitel1.gif," etc.). The applet
11 // assumes that images are in an "images" subdirectory of
12 // the directory in which the applet resides.
13 //
14 // HTML parameter "totalImages" is an integer representing the
15 // total number of images in the animation. The applet assumes
16 // images are numbered from 0 to totalImages - 1.
17
18 import java.applet.Applet;
19 import java.awt.*;
20
21 public class DeitelLoop5 extends Applet implements Runnable {
22     private Image images[];
23     private int totalImages, // total number of images
24         currentImage = 0, // current image subscript
25         sleepTime; // milliseconds to sleep
26     private String imageName; // base name of images
27
28     // The next two objects are for double-buffering
29     private Graphics gContext; // off-screen graphics context
30     private Image buffer; // buffer in which to draw image
31
32     private MediaTracker imageTracker; // used to track images
33
34     private Thread animate; // animation thread
35     private boolean suspended; // toggle on/off
36
37     private int width, height;
38
39     // GUI Components to allow dynamic speed changing
40     private Label sleepLabel;
41     private TextField sleepDisplay;
42     private Panel sleepStuff;
43
44     // load the images when the applet begins executing
45     public void init()
46     {
47         processHTMLParameters();
48
49         if ( totalImages == 0 || imageName == null ) {
50             showStatus( "Invalid parameters" );
51             destroy();
52         }
53
54         images = new Image[ totalImages ];
```

```

55         imageTracker = new MediaTracker( this );
56
57         for ( int i = 0; i < images.length; i++ ) {
58             images[ i ] = getImage( getDocumentBase(),
59                 "images/" + imageName + i + ".gif" );
60
61             // track loading image
62             imageTracker.addImage( images[ i ], 1 );
63         }
64
65         try {
66             imageTracker.waitForID( 0 );
67         }
68         catch( InterruptedException e ) {}
69
70         width = images[ 0 ].getWidth( this );
71         height = images[ 0 ].getHeight( this );
72         resize( width, height + 30 );
73
74         buffer = createImage( width, height );
75         gContext = buffer.getGraphics();
76
77         // set background of buffer to white
78         gContext.setColor( Color.white );
79         gContext.fillRect( 0, 0, 160, 80 );
80
81         setLayout( new BorderLayout() );
82         sleepLabel = new Label( "Sleep time" );
83         sleepDisplay = new TextField( 5 );
84         sleepDisplay.setText( String.valueOf( sleepTime ) );
85         sleepStuff = new Panel();
86         sleepStuff.add( sleepLabel );
87         sleepStuff.add( sleepDisplay );
88         add( "South", sleepStuff );
89     }
90
91     // start the applet
92     public void start()
93     {
94         // always start with 1st image
95         gContext.drawImage( images[ 0 ], 0, 0, this );
96         currentImage = 1;
97
98         // create a new animation thread when user visits page
99         if ( animate == null ) {
100             animate = new Thread( this );
101             animate.start();
102         }
103     }
104
105     // terminate animation thread when user leaves page
106     public void stop()
107     {
108         if ( animate != null ) {
109             animate.stop();

```

```
110         animate = null;
111     }
112 }
113
114 // display the image in the Applet's Graphics context
115 public void paint( Graphics g )
116 {
117     g.drawImage( buffer, 0, 0, this );
118 }
119
120 // override update to eliminate flicker
121 public void update( Graphics g )
122 {
123     paint( g );
124 }
125
126 public void run()
127 {
128     while ( true ) {
129         if ( imageTracker.checkID( currentImage, true ) ) {
130             // clear previous image from buffer
131             gContext.fillRect( 0, 0, 160, 80 );
132
133             // draw new image in buffer
134             gContext.drawImage(
135                 images[ currentImage ], 0, 0, this );
136
137             currentImage = ++currentImage % totalImages;
138         }
139         else // browser fix: help load images
140             postEvent(
141                 new Event( this, Event.MOUSE_ENTER, "" ) );
142
143         try {
144             Thread.sleep( sleepTime );
145         }
146         catch ( InterruptedException e ) {
147             showStatus( e.toString() );
148         }
149
150         repaint(); // display buffered image
151     }
152 }
153
154 public boolean action( Event e, Object o )
155 {
156     try {
157         int i = Integer.parseInt( o.toString() );
158         sleepTime = i;
159     }
160     catch( NumberFormatException nfe ) {
161         showStatus( "Sleep time must be an integer" );
162     }
163
164     return true;
165 }
```

```

165     }
166
167     public boolean mouseDown( Event e, int x, int y )
168     {
169         if ( suspended ) {
170             animate.resume();
171             suspended = false;
172         }
173         else {
174             animate.suspend();
175             suspended = true;
176         }
177
178         return true;
179     }
180
181     public void processHTMLParameters()
182     {
183         String parameter;
184
185         parameter = getParameter( "sleeptime" );
186         sleepTime = ( parameter == null ? 50 :
187                     Integer.parseInt( parameter ) );
188
189         imageName = getParameter( "imagename" );
190
191         parameter = getParameter( "totalimages" );
192         totalImages = ( parameter == null ? 0 :
193                       Integer.parseInt( parameter ) );
194     }
195 }

```



图 14.7 通过 paramHTML 标记来定制 applet

## 14.9 图像映射

图像映射是创建更有趣的网页的一种常用方法。一个图像映射就是一个带有“热点”的图像，用户点击该“热点”可以完成某个操作，比如将另一个不同的网页加载到浏览器中。当用户将鼠标指针移到热点上时，在浏览器的状态栏中通常会显示一个描述信息。这个方法可用于实现一个“指

取帮助” (bubble help) 系统。当用户将鼠标指针移到屏幕的某一个特殊元素上时, 带有“指取帮助”的系统通常会在一个小窗口中显示一条信息, 这个小窗口将自动出现在鼠标指针所指的元素上。在 Java 中, 该信息可以在状态栏中显示。

图 14.8 的程序加载了一个包含很多图标的图像, 该程序的功能是当用户将鼠标指针放在某个图标上时, 将为该图标显示一个描述信息。mouseMove 方法将鼠标的坐标传递给 translateLocation 方法。根据该坐标可以判断出鼠标指针位于哪个图标上。然后, translateLocation 方法就在 applet 的状态栏中显示一个信息, 说明该图标代表什么含义。

在这个 applet 中, 点击鼠标不会产生任何动作。在第 16 章中, 我们将讨论一些将其他网页加载到浏览器中所需的方法。一旦拥有了这些网络技术, 我们就可以修改该 applet, 使每一个图标都与一个不同的 URL 相关联。

```

1 // Fig. 14.8: ImageMap.java
2 // Demonstrating an image map.
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class ImageMap extends Applet {
7     Image mapImage;
8     MediaTracker trackImage;
9     int width, height;
10
11     public void init()
12     {
13         trackImage = new MediaTracker( this );
14         mapImage = getImage( getDocumentBase(), "icons2.gif" );
15         trackImage.addImage( mapImage, 0 );
16
17         try {
18             trackImage.waitForAll();
19         }
20         catch( InterruptedException e ) { }
21
22         width = mapImage.getWidth( this );
23         height = mapImage.getHeight( this );
24         resize( width, height );
25     }
26
27     public void paint( Graphics g )
28     {
29         g.drawImage( mapImage, 0, 0, this );
30     }
31
32     public boolean mouseMove( Event e, int x, int y )
33     {
34         showStatus( translateLocation( x, y ) );
35         return true;
36     }
37
38     public boolean mouseExit( Event e, int x, int y )
39     {
40         showStatus( "Pointer outside ImageMap applet" );
41         return true;
42     }
43
44     public String translateLocation( int x, int y )

```

```

45  {
46      // determine width of each icon (there are 6)
47      int iconWidth = width / 6;
48
49      if ( x >= 0 && x <= iconWidth)
50          return "Common Programming Error";
51      else if ( x > iconWidth && x <= iconWidth * 2 )
52          return "Good Programming Practice";
53      else if ( x > iconWidth * 2 && x <= iconWidth * 3 )
54          return "Performance Tip";
55      else if ( x > iconWidth * 3 && x <= iconWidth * 4 )
56          return "Portability Tip";
57      else if ( x > iconWidth * 4 && x <= iconWidth * 5 )
58          return "Software Engineering Observation";
59      else if ( x > iconWidth * 5 && x <= iconWidth * 6 )
60          return "Testing and Debugging Tip";
61
62      return "";
63  }
64  }

```



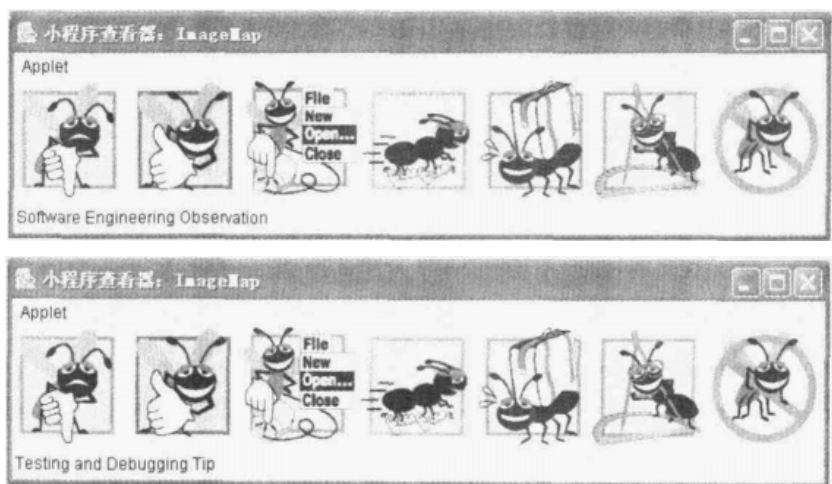


图 14.8 说明一个图像映射

## 小结

- Applet 类的 `getImage` 方法用于将一个图像加载到 applet 中。
- Applet 类的 `getDocumentBase` 方法用于以 URL 类对象的方式返回 HTML 文件（引用了 applet）在 Internet 上的位置。
- URL 即统一资源定位器，这是表示 Internet 上信息地址的一种标准格式。
- Java 目前支持两种图像格式，即 GIF 和 JPEG，在文件名中分别使用后缀 `.gif` 或 `.jpg` 表示这两种格式。
- 当引用 `getImage` 方法时，它会启动一个加载图像（或从 Internet 上下载图像）的独立线程。这样，在加载图像的同时，程序仍然可以继续执行。
- Graphics 的 `drawImage` 方法用于在一个 applet 上显示图像。
- ImageObserver 是由 Component 类提供的一个接口。当一个图像的其余部分都加载完时，系统会自动通知 ImageObserver 更新已显示的图像。
- Image 的 `getWidth` 方法和 `getHeight` 方法都带有一个 ImageObserver 参数，分别用于确定图像的宽度和高度。这样，随着 Image 的加载，可以不断进行更新。
- 显示动画的每一幅图像时，通常会发生闪烁，这是由于调用了 applet 的 `update` 方法。`update` 方法清除 applet 的过程，是用当前背景的颜色绘制一个充满 applet 的填充矩形，这样就可以把上一次绘制的图像覆盖，因此导致了闪烁现象的发生。
- 每次调用 `repaint` 方法时，都将启动一个独立的线程（它会创建一个独立的方法调用堆栈）来完成 applet 的重画。
- 采用图形双缓存技术后，当程序向屏幕上显示一幅图像时，就可以同时在屏幕外的一个缓冲区中创建下一幅图像。这样，到显示下一幅图像时，就可以流畅地把它放到屏幕上了。
- 在需要使用除 `paint` 之外的其他方法来进行绘图的程序中，图形双缓存是很有用的。屏幕外的缓冲区可以在方法之间，甚至在不同类之间进行传递，以便使其他的方法或对象能够在该屏幕外的缓冲区上进行绘图。
- GIF 图像可以使用交错格式和非交错格式来保存。当显示一个非交错格式的图像时，图像会

随着像素信息的读取而从上到下逐块地出现。当显示交错格式的图像时,首先会显示第一批像素行,使图像的大致轮廓出现;然后再显示第二批像素行,将图像进行细化;依次类推,使整个图像逐渐清晰,直到全部显示出来。

- 实现一个图形双缓存有两个关键点——一个是 Image 引用,一个是 Graphics 引用。Image 用于保存要显示的实际像素,Graphics 用于绘制像素。
- 每个图像都有一个相关的图形环境——即一个 Graphics 类的对象,用于完成绘图。
- 图形双缓存所用的 Image 和 Graphics 引用通常称为屏外图像和屏外图形环境,因为它们实际上操作的并非屏幕上的像素。
- Applet 类的 createImage 方法用于创建一个空图像,它是通过实例化一个 Image 的子类对象来完成的,该方法带有两个整型参数——空图像的宽度和高度。
- Image 的 getGraphics 方法用于获取与一个 Image 对象相关的图形环境(即 Graphics 对象)。
- 可以利用 MediaTracker 对象来登记图像,使程序能够判断图像何时加载完。MediaTracker 还提供了下面的功能:在程序继续执行之前等待一个或几个图像加载,以及确定加载图像时是否产生了错误。
- MediaTracker 构造函数带有一个参数——一个 ImageObserver(图像将绘制在该对象上)。
- MediaTracker 的 addImage 方法可以通过一个 MediaTracker 对象来登记一个图像。第一个参数是一个 Image 引用。第二个参数是一个整数 ID,用于标识该图像。如果想要将多个图像作为一个组来处理,只需为它们设置相同的 ID 值。
- MediaTracker 的 waitForID 方法可用于强制程序进行等待,直到由其整型参数指定的图像加载完。MediaTracker 类的 waitForAll 方法可用于阻塞程序的执行,直到所有登记的图像都加载完。
- 为了消除所有的闪烁,需要重写 Applet 类的 update 方法。在该方法中调用 paint 方法,但不会清除 applet。
- 为了可以“友好地浏览”动画,应在一个独立的线程上运行动画。在 Applet 类的 start 方法中创建一个 Thread,并启动动画。在 Applet 类的 stop 方法中停止线程,并终止动画(当用户移到另一个网页时调用 stop 方法)。Thread 的 suspend 方法和 resume 方法分别用于暂停和恢复 applet。
- Java 提供了两种在 applet 中播放音频的机制——Applet 类的 play 方法及 AudioClip 接口的 play 方法。
- 如果要在程序中将播放一段声音,可以使用 Applet 类的 play 方法将声音加载进来,然后播放一次;当这段声音播放完后,就对其设置无用单元回收处理标记。
- Applet 类的 getDocumentBase 方法用于指示将加载 applet 的 HTML 文件放在什么位置。
- Applet 类的 getCodeBase 方法用于指示将 applet 的 .class 文件放在什么位置。
- HTML 的 param 标记用于指定 applet 的一个参数。param 标记行必须出现在 applet 的开始标记和结束标记之间,其每个参数都有一个 name 和一个 value。
- Applet 类的 getParameter 方法用于获取与某个参数相关的 value 值,并将该 value 作为一个字符串返回。如果没有包含指定参数的 param 标记行,那么 getParameter 将返回 null。
- 一个图像映射就是一个带有“热点”的图像,用户点击该“热点”可以完成某个操作,比如将另一个不同的网页加载到浏览器中。



## 术语

addImage method	addImage 方法	graphics double buffering	图形双缓存
animation	动画	image	图像
animation loop	动画循环	Image class	Image 类
animation speed	动画速度	ImageObserver class	ImageObserver 类
.au file format for sound files	声音文件的.au 文件格式	imageUpdate method of Applet class	Applet 类的 imageUpdate 方法
audio	音频	JPEG image format	JPEG 图像格式
AudioClip class	AudioClip 类	load an audio clip	加载一个音频剪辑
audio clips	音频剪辑	load an image	加载一个图像
checkID period	checkID 周期	loop method of AudioClip class	AudioClip 类的 loop 方法
display an image	显示一个图像	loop on an audio clip	在一个音频剪辑上循环
double buffering to reduce flicker	利用双缓存来减少闪烁	MediaTracker class	MediaTracker 类
drawImage method of Graphics class	Graphics 类的 drawImage 方法	multimedia	多媒体
fetch an image from a server	从服务器上获取图像	play method of AudioClip class	AudioClip 类的 play 方法
flicker	闪烁	play method of Applet class	Applet 类的 play 方法
flicker reduction	闪烁减少	restart an animation	重新启动一个动画
frame of an animation	动画帧	smooth animation	流畅的动画
frames order	帧的顺序	sound	声音
getAudioClip method	getAudioClip 方法	sound clip	声音片段
getCodeBase method	getCodeBase 方法	sound loop	声音循环
getDocumentBase method	getDocumentBase 方法	sound track	声音轨迹
getGraphics method	getGraphics 方法	stop an animation	停止动画
getImage method of Applet class	Applet 类的 getImage 方法	stop method of AudioClip class	AudioClip 类的 stop 方法
getParameter method	getParameter 方法	stop an audio clip	停止一个音频剪辑
.gif image file	.gif 图像文件	URL of an image	一个图像的 URL
graphics	图形	waitForAll method	waitForAll 方法
Graphics class	Graphics 类	waitForID method	waitForID 方法
graphics context	图形环境		

## 自测练习

### 14.1 填空：

- Applet 类的 \_\_\_\_\_ 方法可用于加载一个图像到 applet 中。
- Applet 类的 \_\_\_\_\_ 方法可用于以 URL 类对象的方式返回 HTML 文件（引用了 applet）在 Internet 上的位置。

- e) 表示 Internet 上信息地址的一种标准格式是 \_\_\_\_\_。
  - d) Graphics 的 \_\_\_\_\_ 方法用于在一个 applet 上显示图像。
  - e) 在显示动画的每一幅图像时, 通常会发生闪烁。这是由于调用了 applet 的 \_\_\_\_\_ 方法。利用该方法清除 applet 的过程, 是使用当前的背景色绘制一个充满 applet 的填充矩形, 这样就会把刚绘制的图像覆盖, 因此导致了闪烁。
  - f) 采用 \_\_\_\_\_ 技术后, 当程序向屏幕上显示一幅图像时, 就可以同时在屏幕外的一个缓冲区中创建下一幅图像。这样, 到显示下一幅图像时, 就可以流畅地将其放到屏幕上。
  - g) 当显示一个 \_\_\_\_\_ 格式的图像时, 它会先显示第一批像素行, 使图像的大致轮廓出现; 然后再显示第二批像素行, 将图像进行细化; 依次类推, 使整个图像逐渐清晰, 直到全部显示出来。
  - h) 实现一个图形双缓存有两个关键点——一个是 \_\_\_\_\_ 引用, 一个是 \_\_\_\_\_ 引用。第一个用于保存要显示的实际像素; 第二个用于绘制像素。
  - i) 可以利用 \_\_\_\_\_ 对象来登记图像, 使程序能够判断图像何时加载完。
  - j) Java 提供了两种在 applet 中播放音频的机制——Applet 类的 play 方法及 \_\_\_\_\_ 接口的 play 方法。
  - k) \_\_\_\_\_ 是带有“热点”的图像, 用户点击“热点”就可以完成某一操作, 例如将另一个不同的网页加载到浏览器中。
- 14.2 判断下列句子是否正确。如果不正确, 请解释原因。
- a) Java 目前支持两种图像格式。在文件名中分别用后缀 .jif 或 .gpg 来表示这两种格式。
  - b) 将 Applet 类的 update 方法重写, 调用 paint 方法但不清除 applet, 这样可以显著地减少动画的闪烁。
  - c) 在一段音频播放完后, 就立刻对其进行无用单元回收处理。
  - d) Java 的 param 标记用于指定 applet 的一个参数 param 标记行必须出现在 applet 的开始标记和结束标记之间。每个参数都有一个 name 和一个 value。

## 自测练习答案

- 14.1 a) getImage。 b) getDocumentBase。 c) URL。 d) drawImage。 e) update。 f) 图形双缓存。  
g) 交错。 h) Image, Graphics。 i) MediaTracker。 j) AudioClip。 k) 图像映射。
- 14.2 a) 不正确。应该是 .gif 和 .jpg。  
b) 正确。  
c) 不正确。只是对这段音频设置无用单元回收处理标记, 当无用单元回收处理程序能够运行时才对其进行无用单元回收处理, 不一定是立刻就处理。  
d) 不正确。param 标记是 HTML 的, 不是 Java 的。

## 练习

- 14.3 描述一下如何使一个动画可以“友好地浏览”。
- 14.4 讨论 Java 中消除闪烁的几种方法。

- 14.5 解释图形双缓存技术。
- 14.6 描述Java的播放和操作音频剪辑的方法。
- 14.7 如何用一个HTML文件的信息来定制Java applet?
- 14.8 怎样使用MediaTracker对象?使用MediaTracker时应注意些什么?
- 14.9 解释一下如何使用图像映射。列出10个使用图像映射的应用程序。
- 14.10 (动画) 创建一个通用的Java动画applet。在该applet中,应允许用户指定要显示的帧序列、图像显示的速度、以及动画运行时应播放的音频等。
- 14.11 (讲故事) 将大量的名词、动词、冠词、介词等的音频记录下来。然后使用随机数产生器来构成句子,并让程序将这些句子读出来。
- 14.12 (五行打油诗) 修改练习8.10的五行打油诗编写程序,将所创建的五行打油诗读出来。
- 14.13 (屏幕保护程序) 利用一系列图像所构成的动画来创建一个屏幕保护程序。构造几种特殊效果:分解图像,旋转图像,使图像淡入和淡出,将图像移出屏幕的边界等。
- 14.14 (随机地擦除一个图像) 假设在矩形屏幕区域中显示图像。擦除该图像的一种方法是同时将每个像素都设置成同一个颜色,但这样产生的效果很单调。编写一个Java applet,首先显示一个图像,然后使用随机数产生器来选择单个像素进行擦除。当图像的大部分都已擦除后,就将所有剩余的像素一次擦除。可以逐行地选择个别像素。可以试用几种不同的方式来实现这个任务。例如,可以利用随机的直线或图形来擦除屏幕的区域。
- 14.15 (随机的内部图像转换) 这个程序可以产生一种良好的视觉效果。如果在屏幕的某一给定区域显示着一幅图像,而你想在该区域显示另一幅图像,那么就要先将新的屏幕图像保存在一个屏外缓冲区中,然后随机地将新图像的像素复制到给定的显示区上,覆盖原来的像素点。当绝大部分像素都已复制完后,就将整个新图像复制到给定的显示区上,以保证所显示的新图像的完整性。为了实现该程序,可能需要使用PixelGrabber类和MemoryImageSource类(参见Java API文档中有关这些类的描述)。可以试用几种不同的方式来实现这个任务。例如,可以随机选择新图像上的一条直线或一个图形,使用该直线或图形的像素来覆盖原来图像对应位置的像素。
- 14.16 (背景音乐) 为应用程序添加背景音乐。在与该应用程序进行正常交互操作的同时,使用AudioClip类的loop方法在背景中播放音频。
- 14.17 (项目:多媒体健美操) 开发一个健美操练习的Java applet,采用适当的背景音乐、声音及鼓励性的语言指导,以动画的方式显示健美操练习的各个步骤。为使这个程序更加国际化,可以提供几种不同的语言指导,并使播放的音乐适合于每个地区。允许用户自己制定他们的练习程序,以适合特殊的练习需要。允许用户按初级、中级和高级的标准来制定自己的练习程序。另外,还可以增加其他的功能。
- 14.18 (项目:多媒体创建系统) 开发一个通用的多媒体创建系统。程序中应允许用户构造多媒体环境,其中可以包含文本、音频、图像、动画,最终还可以加入视频。还应允许用户从屏幕上显示的一个目录中选择所需的多媒体元素,并根据这些元素编排多媒体环境。程序中应提供一些组件,使用户可以动态地调整多媒体环境。
- 14.19 (视频游戏) 开发一个Java视频游戏程序。和同学们比一比,看谁能开发出最好的、有独创性的视频游戏。
- 14.20 (滚动的选取框标志) 创建一个Java applet,从右向左(或从左向右,以适合语言顺序)地滚动屏幕上的字符,跨过一个类似选取框的显示标志。建议连续不断地循环显示文

本, 这样, 当文本从显示标志的一边移出时, 它会在另一边重新显示出来。

- 14.21 (文本闪烁程序) 创建一个 Java applet, 反复闪烁屏幕上的文本。利用一个只含有纯背景颜色的图像来点缀文本, 以达到闪烁的效果。允许用户控制“闪烁速度”及背景颜色或模式。
- 14.22 (图像闪烁程序) 创建一个 Java applet, 反复闪烁屏幕上的图像。利用一个只含有纯背景颜色的图形来点缀图像, 以达到闪烁的效果。
- 14.23 (滚动的图像选取框) 创建一个 Java applet, 滚动一个图像, 跨过一个选取框屏幕。
- 14.24 (物理演示: 反弹球) 开发一个动画 applet, 显示一个反弹的小球。给小球设置一个固定的水平速率。允许用户指定复原系数, 若复原系数为 75%, 表示小球反弹后只能达到反弹前高度的 75%。在这个演示中应考虑重力影响, 这样, 反弹小球的轨迹就应该是一个抛物线。每次当小球撞击地面时, 就播放类似弹簧反弹的声音。
- 14.25 (摆动) 开发一个动画 applet, 显示一个摆动的钟摆。允许用户指定一个阻尼因子, 使摆动可以逐渐减慢, 直到最终停止。
- 14.26 (项目: 飞行模拟器) 开发一个飞行模拟器 applet。这是一个富有挑战性的项目, 也是一个能够与同学进行比赛的项目。
- 14.27 (汉诺塔) 编写一个程序, 以动画方式演示练习 4.38 中汉诺塔问题的解决过程。当从柱子上拿起一个盘子或滑落一个盘子时, 就播放“嗖”的背景声音。当一个盘子落在柱子上时, 就播放“咣当”的背景声音, 并播放一些合适的背景音乐。
- 14.28 (龟兔赛跑) 编写一个程序, 以动画方式模仿练习 5.41 中的龟兔赛跑问题。可以加上一些讲解员的画外音, 例如“比赛双方站到起跑线上”, “它们跑出去了!”, “兔子跑到了前面”, “乌龟在努力地赶上来”, 等等。在比赛进行过程中, 播放适当的背景声音。还要播放一些模仿动物赛跑的声音, 不要忘了观众的欢呼声! 将这场比赛放在一个山脚下的场景中举行。
- 14.29 (指取帮助) 利用一个图像映射来实现“指取帮助”功能。当鼠标指针移到一个给定的图像上时, 就显示相应的文本, 以帮助用户了解该图像的功能。可能需要加入一些延迟, 这样, 只有当鼠标指针在一个图像上停留一会儿时, 才显示帮助信息。
- 14.30 (数字时钟) 实现一个 applet, 在屏幕上显示一个数字时钟。可以添加一些选项来完成下列功能: 控制时钟的大小, 显示日期、月份和年份, 发出闹铃声, 在指定的时刻播放某种声音等。
- 14.31 (模拟时钟) 创建一个 Java applet, 显示一个带有时针、分针和秒针的模拟时钟, 各个表针应随着时间的推移而进行相应的移动。
- 14.32 (动态的股票证券行情软件) 创建一个 Java applet, 它可以阅读一个描述投资者股票证券行情的文件。对于投资者持有的每种股票, 该文件都包含了相应的股票行情符号及投资者所持有的股数。然后, applet 再访问 Internet 上的一些股票行情服务机构, 将那些与投资者股票有关的股票交易信息筛选出来。当 applet 获取了新的股票价格时, 就在屏幕上显示一个电子表格, 并动态地更新该表格。在这个电子表格中, 应显示每一种股票的符号、该股票的最新价格、股数以及该股票的最新总值。该电子表格还应统计投资者拥有的所有证券的最新总值。投资者应该能够一边进行其他工作, 一边在屏幕的一角运行这个 Java applet。
- 14.33 (动态地定制新闻便笺) 学习完第 16 章后, 读者就会了解如何开发一个能够访问 WWW 的、基于 Internet 的 Java 应用程序。开发一个“未来报纸”, 用户可以利用图形用户界

面来设计一个特制的动态报纸，以满足用户对特殊信息的需要。然后，定期地或连续不断地从 WWW 上收集信息，你会惊奇地看到，在 WWW 上竟有如此多的免费电子出版物。

- 14.34 (动态声音及图形万花筒) 开发一个万花筒 applet，模仿流行的儿童玩具万花筒，显示反射的图形。加入音响效果，与图形的动态改变相匹配。
- 14.35 (动态的拼图游戏产生器) 创建一个 Java 拼图游戏产生器和操作器。由用户指定一幅图像，applet 先将该图像加载并显示出来，然后再把它随机地分割成各种形状并打乱。用户可以使用鼠标随意移动这些拼图，直到把整张图拼出来。当打乱图像及将其拼回原位时，可以加入不同的声音效果。可以监视每一块拼图及其原来所在的位置，然后利用声音效果帮助用户将拼图放到正确的位置。
- 14.36 (教魔术) 如果读者曾尝试过变魔术，就会知道这有多么困难。尤其是从书上学习就更加难，因为书中不可能把变魔术的各个动作都演示出来。开发一个多媒体的 Java applet，帮助人们学会如何变魔术。尤其要演示变魔术的各个细节。在开始编写这个 applet 之前，可能需要先阅读一本有关魔术的书籍。
- 14.37 (迷宫生成器和走迷宫) 开发一个多媒体的迷宫生成器和行走器（可基于练习 5.38、练习 5.39 和练习 5.40 的迷宫程序开发项目）。用户可以通过指定迷宫的行数、列数以及难度来定制迷宫。让一个动画的鼠标在迷宫中行走，利用声音来表演鼠标的移动。
- 14.38 (骑士旅行) 将练习 5.22 和练习 5.23 的骑士旅行程序改写成多媒体形式。
- 14.39 (弹球机) 这又是一个富有挑战性的问题。开发一个 Java applet，模拟弹球机。和同学们比一比，看谁能开发出最好的、具有独创性的多媒体弹球机。充分利用你所能想到的各种多媒体手段，使这个弹球游戏更有趣，尽量使游戏规则接近真正的弹球游戏。
- 14.40 (转盘赌博) 研究一下转盘赌博的规则，开发一个多媒体的转盘赌博游戏。创建一个动画的旋转转盘。通过音频来模拟小球在每个数字小格之间跳动的声音，以及小球最终落入某个格子的声音。在转盘的旋转过程中，各玩家可以下赌注。当小球最终落入某个格子时，就根据每个玩家的输赢来更新他们各自的银行账户余额。
- 14.41 (掷骰子赌博) 模拟掷骰子赌博的整个过程。利用一个图形表示掷骰子的桌子。允许多个玩家下赌注。利用动画来表示玩家掷骰子以及骰子不断旋转直到最终停下的过程，利用声音来模仿桌子周围喋喋不休的声音。每玩完一把，就根据每个玩家所下的赌注来更新他们各自的银行账户余额。
- 14.42 (老虎机) 开发一个多媒体的老虎机。老虎机有三个转轮，在每个转轮上放置不同的奖品和符号。利用随机数产生器来模拟每个转轮的旋转和停止。
- 14.43 (赛马) 创建一个 Java 程序来模拟赛马过程。赛马程序有多个竞赛者，利用音频来模拟比赛讲解员的讲解，播放适当的声音来指示每个骑手的正确比赛状态，并宣布最终结果。可以试试模拟经常在狂欢节上举行的那种赛马。玩家轮流使用鼠标操作，而且必须使用鼠标完成一些技巧性操作来提高马匹的能力。
- 14.44 (弹子游戏) 创建一个多媒体的弹子游戏。每个玩家轮流用鼠标操作弹子杆，在相应的角度击中弹子球，使弹子球落入槽中。该 applet 应有计分功能。
- 14.45 (时装设计程序) 开发一个多媒体的时装设计工具，帮助时装设计师设计出流行的时装。该工具应允许设计者通过选择颜色、式样、装饰特征等来设计服装。
- 14.46 (画家) 设计一个进行艺术设计的 Java applet，画家可以利用该程序所提供的大量绘图，使用图像和动画等类似功能来创建一个动态的多媒体艺术作品。

- 14.47 (烟火设计程序) 创建一个 Java applet, 用于设计一个烟火效果。创建一些烟火表演, 然后为这些烟火配上声音, 以达到最佳效果。
- 14.48 (房间设计程序) 开发一个 Java applet, 帮助用户布置家中的家具。添加一些功能, 帮助用户找到最佳的布置效果。
- 14.49 (填字谜游戏) 开发一个多媒体的填字谜游戏。程序中应使玩者能很容易地填充和擦除字母。将该程序与一个大型的电子词典连接起来。这个 applet 应能根据已填入的字母提示单词。再提供一些其他的功能, 使填字谜游戏迷们更容易操作。
- 14.50 (合成乐器) 开发一个多媒体的 Java applet, 模仿一个合成乐器。该 applet 应显示一个带有各种声音选项的键盘。用户通过按键来演奏音乐。用户可以选择各种乐器的声音: 钢琴、单簧管、鼓、钹等。
- 14.51 (自动钢琴) 创建一个 Java applet, 模仿自动钢琴。该 applet 接收一个保存在文件中的活页乐谱, 在这个乐谱中指明应演奏哪个琴键或哪几个琴键, 以及持续的时间。可以事先将钢琴的 88 个琴键的音调记录下来。
- 14.52 (活页乐谱生成器/自动演奏) 编写一个 Java applet, 为上个练习中所创建的文件显示活页乐谱。
- 14.53 (音乐教师) 开发一个 Java applet, 帮助学生学习弹钢琴。在 applet 中应显示钢琴的键盘, 并使用各种视觉和声音效果。
- 14.54 (算术家庭教师) 将练习 4.31、练习 4.32 和练习 4.33 中开发的计算机辅助教学系统(CAI) 改写成多媒体版本。
- 14.55 (卡拉 OK) 创建一个卡拉 OK 系统, 播放歌曲的配乐, 并适时地显示歌词。
- 14.56 (使图像引人注目) 如果想要将某一幅图像着重显示, 可在该图像周围放一圈模拟的小灯泡。可以让这些小灯泡同时闪亮, 也可以让它们依次闪亮。
- 14.57 (物理演示: 动力学) 开发一个 Java applet, 演示下面一些物理概念: 能量, 惯性, 动量, 速度, 加速度, 摩擦力, 弹性系数, 重力, 等等。加入一些视觉和声音效果。
- 14.58 (在线的产品目录) 开发一个在线的多媒体目录, 消费者可以从中选择所需的产品。学习完第 16 章后, 就能够处理一些有关网络的问题了。
- 14.59 (反应时间/反应精确度测试程序) 创建一个 Java applet, 在屏幕上随机地移动一个图形, 用户使用鼠标在屏幕上捕捉这个图形并点击它, 这样该图形的移动速度和大小可以发生变化。对于某一个给定大小的图形, 统计一下用户平均需花费多少时间才能捕捉住该图形。图形越小, 移动的速度越快, 用户就越难捕捉到它。
- 14.60 (图像缩放) 创建一个 Java applet, 用于将图像移近或移远(即放大或缩小)。
- 14.61 (日历/备忘录文件) 使用声音和图像创建一个通常的日历和备忘录文件。例如, 当某天是读者的生日时, 就应该播放“生日快乐”歌。该 applet 应显示与一些重要事件相关的图像并播放相应的声音。另外, 在重要事件之前还应有预先提醒的功能。例如, 最好提前一周提醒, 以便能有时间去选择一张合适的贺卡。
- 14.62 (旋转图像) 创建一个 Java applet, 可以将图像旋转某个角度(最大为 360 度)。作为一个选项, 该 applet 应允许用户指定将图像进行连续旋转, 并可动态地调整旋转速度。
- 14.63 (为黑白效果的照片和图像涂上颜色) 创建一个 Java applet, 可以给黑白效果的照片和图像涂上颜色。提供一个调色板, 在图像的不同区域应使用不同的颜色。
- 14.64 (项目: 自动柜员机) 开发一个 Java 应用程序, 实现一个自动柜员机, 并模拟它与一个银行账户(由另一台计算机维护)的交互操作。这个 applet 的第一个版本应完全按照

实际的操作过程模仿自动柜员机。然后发挥读者创造力，设计一个“未来自动柜员机”。使用图形、动画、声音及Java的任何功能，还可以联系WWW和Internet。[注意：这个项目需要使用第14章、第15章和第16章中的高级Java技术。建议用户现在尽量使用已在第9章到第14章中学过的Java图形、GUI、多线程和多媒体技术，然后在学完第15章后加入文件处理功能，学完第16章后再加入客户/服务器网络功能，这是一个很好的协作项目。]

- 14.65 (多媒体的 Simpletron 模拟程序) 修改在前面章节练习中开发的 Simpletron 模拟程序，增加多媒体功能。加入类似计算机的声音，表示 Simpletron 在执行指令。当发生致命错误时，加入一个打碎玻璃的声音。利用闪烁的灯光表示当前正在操作哪一个内存单元或哪一个寄存器。还可使用其他的一些多媒体技术，使这个模拟程序成为更有价值的教学工具。

# 第15章 文件和流

## 教学目标

- 学会创建、读、写和更新文件
- 理解 Java 流类的层次关系
- 学会使用 FileInputStream 和 FileOutputStream 类
- 学会使用 DataInputStream 和 DataOutputStream 类
- 熟悉顺序访问和随机访问文件的操作
- 创建一个对随机访问文件进行交易处理的程序
- 学会使用 File 类

## 15.1 简介

使用变量和数组存储数据是临时性的，例如，当局部变量超出它的定义范围或者当程序终止后，这些数据就会丢失。而文件能够长期保存大量的数据，即使在创建数据的程序终止后，数据仍然存在。保存在文件中的数据通常称为“永久数据”，计算机利用诸如磁盘、光盘和磁带这样的二级存储设备来存储文件。本章将讲述如何使用 Java 程序来创建、更新和访问数据文件，这里既讨论了随机访问文件，也讨论了顺序访问文件，并且将指出对于每种文件来说最适合它的应用程序。

商业应用需要访问大量的永久数据，因此对于支持商业应用的语言来说，文件操作是最重要的一项功能。在本章，我们将讨论 Java 强大而丰富的文件操作以及流的输入输出特性。Java 强大的流处理功能中所包含的文件处理功能，使得程序能够在内存、文件中或通过网络读写数据。本章中将着重讨论两个内容，第一是介绍文件处理范例，另外就是提供具有流处理功能的阅读器，以支持将在下一章介绍的网络特性。

### 软件工程视点 15.1

对于那些在 WWW 的任何地点都能够访问到的 Java applet，如果让它们能够读写客户系统上的文件，则是非常危险的。因此，文件访问程序通常是作为 Java 的应用程序来实现的。

## 15.2 数据组织

归根到底，一台计算机能操作的数据项就是“0”和“1”的组合，因为我们可以很容易地让一种电子设备表示两种稳定状态——一个状态代表“0”而另一个状态代表“1”。计算机的另一个显著特点就是计算机的操作只包括对 0 和 1 的基本处理。

计算机中最小的数据项是能够表示 0 和 1 的数据项。这样的数据项称为一个“位”（bit）。计算机电路可以进行简单的位操作，诸如检查某位的值，设置某位的值，反转某位（从 1 到 0，或者从 0 到 1）。

对于程序员来说，使用这种以底层位表示的数据来编写程序是非常麻烦的。因此，程序员宁愿



采用其他形式的数据,比如十进制数字(0~9)、字母(A~Z, a~z)和特殊符号(\$、@、%、&、\*、()、-、+、”、?、/)。数字、字母和特殊符号称为字符,在计算机中用来编程和表示数据的所有字符的集合称为计算机的字符集。

由于计算机只能对0和1操作,因此计算机字符集中的每个字符都用0和1的模式来表示(Java中的字符是双字节的标准字符,即Unicode字符,字节通常由8个位组成)。程序员利用字符来创建文件和数据项,而计算机则按照位模式来处理这些字符。

正如字节是由位组成的,域(field)则是由字符组成的:一个域是表达一定含义的一组字符。例如,我们可以通过只由大、小写字母组成的域来表示一个人的名字。

随着从位到字符,再到域的一步发展,计算机操作的数据形成了一个数据层次(data hierarchy),其中的数据项变得越来越大,结构也越来越复杂。

一条记录(record,即Java中的class)由几个域(在Java中称为实例变量)组成。例如,在一个工资系统中,雇员的记录包含以下几个域:

1. 雇员标识号
2. 姓名
3. 地址
4. 每小时工资
5. 免税申请号
6. 年薪
7. 联邦税收额,等等

这样,一条记录就是一组相关的域。在上面的这个例子中,每个域都属于同一个雇员。当然,一个公司会有很多雇员,而每个雇员都会有一条工资记录。一个文件就是一组相关记录。正常情况下,一个公司的工资文件会包含每个雇员的记录。这样,一个小公司的工资文件可能会有22条记录,而一个大公司的工资文件则可能多达10 000条记录。一个公司有许多文件,并且每个文件都包含几百万个字符,这种情况也是很常见的。图15.1中显示了数据层次。

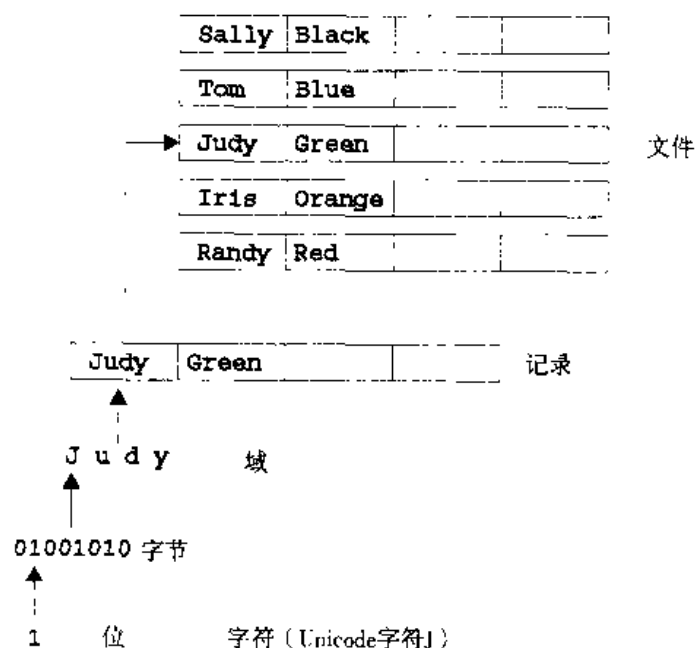


图 15.1 数据层次

为了更容易地从文件中检索数据,每条记录至少要有个域作为记录关键字。一个记录关键字说明一个记录属于某个人或某个实体,从而将该记录同文件中的其他记录区分开来。在上面定义的工资记录中,雇员标识号一般作为记录关键字。

文件中有许多组织记录的方法。最普通的组织类型称为顺序文件,这种文件最典型的方法是按照记录关键字域来存储记录。在工资文件中,记录通常按照雇员标识号的顺序存放。在文件中,第一个雇员的记录包含最小的雇员标识号,随后的记录包括递增的更大的雇员标识号。

许多企业利用很多互不相同的文件来存储数据。例如,很多公司可能会有工资文件、应收账款文件(列出客户应支付的钱款)、应付账款文件(列出向供应商支付的钱款)、明细文件(列出企业管理所需要的各种资料),以及其他许多类型的文件。一个相关文件组有时称为一个数据库。一个用来创建和管理数据库的程序集称为数据库管理系统(DBMS)。

### 15.3 文件和流

Java 简单地把每个文件都视为一个顺序字节流(如图 15.2 所示)。每个文件或者结束于一个文件结束标志,或者根据系统维护或管理数据结构中所记录的具体字节数来终止。当打开一个文件时,就将创建一个对象,同时会有一个流和该对象关联。当我们开始运行一个 Java 应用程序或者 Java applet 时,就自动创建了 3 个流对象——`System.in`、`System.out`、`System.err`。与这些对象相关联的流提供了一个程序和某个文件或者设备之间的通信通道。例如,`System.in` 对象(标准输入流对象)能够让一个程序接收键盘输入的字节,`System.out` 对象(标准输出流对象)能够使一个程序向屏幕输出数据,`System.err` 对象(标准错误流对象)能够使一个程序向屏幕输出错误信息。

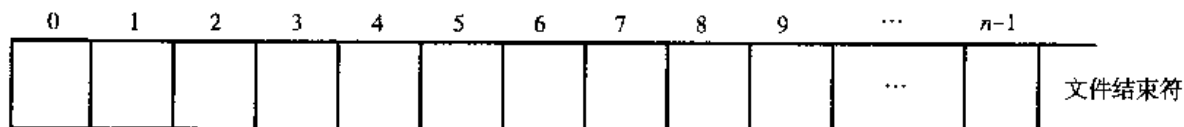


图 15.2 以 Java 的观点来查看一个  $n$  个字节的文件

为了在 Java 中进行文件操作,我们必须引入 `java.io` 软件包。这个软件包中包括 `FileInputStream` (从一个文件输入)和 `FileOutputStream` (输出到一个文件)这样的流类。通过创建从 `InputStream` 类和 `OutputStream` 类派生(即继承它们的功能)出的这些流类的对象,可以分别打开文件。这样,这些流类的方法都能适用于文件流。为了完成原始数据类型的输入和输出,`DataInputStream` 类和 `DataOutputStream` 类的对象需要和文件流类一起使用。图 15.3 总结了 Java 输入/输出类的这种继承关系,下面的讨论概述了图 15.3 所示的每种类的功能。

Java 为输入/输出操作提供了许多类,在这一节我们将概述每种类并解释它们是怎样同其他类相关联的。在这一章的剩余部分,我们将应用几个关键的流类,以完成包括对顺序访问文件和随机访问文件进行创建、操作和删除等多种文件处理的程序。我们还会给出一个有关 `File` 类的详细例子,它对于我们获得有关文件和目录的信息很有帮助。在第 16 章,我们将进一步使用流类来完成网络应用。

`InputStream` (`Object` 的子类)和 `OutputStream` (`Object` 的子类)是抽象类,它们分别为输入和输出操作定义了方法,抽象类的派生类也重载了这些方法。

文件的输入和输出由 `FileInputStream` (`InputStream` 的子类)和 `FileOutputStream` (`OutputStream` 的子类)来完成。我们将在这一章的例子中进一步使用这些类。

## Java 流类的继承关系

```

Object
├── File
├── FileDescriptor
├── StreamTokenizer
├── InputStream
│   ├── ByteArrayInputStream
│   ├── SequenceInputStream
│   ├── StringBufferInputStream
│   ├── PipedInputStream
│   ├── FileInputStream
│   ├── FilterInputStream
│   │   ├── DataInputStream
│   │   ├── BufferedInputStream
│   │   ├── PushBackInputStream
│   │   └──LineNumberInputStream
│   └── OutputStream
│       ├── ByteArrayOutputStream
│       ├── PipedOutputStream
│       ├── FileOutputStream
│       ├── FilterOutputStream
│       │   ├── DataOutputStream
│       │   ├── BufferedOutputStream
│       │   └── PrintStream
│       └── RandomAccessFile

```

图 15.3 Java.io 软件包的类继承关系

管道 (pipe) 是线程间的同步通信通道, 两个线程间就可以建立一个管道。一个线程通过 `PipeOutputStream` 类 (`OutputStream` 的子类) 来向另一个线程传送数据, 目标线程通过 `PipeInputStream` 类 (`InputStream` 的子类) 从管道中读取信息。

`PrintStream` 类 (`FilterOutputStream` 的子类) 用来向屏幕 (或者由本地计算机中的操作系统所定义的“标准输出”) 输出。事实上, 我们就是采用 `PrintStream` 类来进行文本输出的; `System.out` 就是一个 `PrintStream` (`System.err` 同样如此)。

`FilterInputStream` 过滤了 `InputStream`, 而 `FilterOutputStream` 过滤了 `OutputStream`。简单地说, 过滤操作意味着过滤流提供了一些额外的功能, 诸如缓冲、监视行号或者将数据字节集中到原始数据类型的单元中。

以原始字节的方式读取数据虽然速度很快, 但并不方便。通常情况下, 程序想以字节集合的形式 (例如整数、浮点数、双浮点数等) 来读取数据。为了达到这个目的, 我们需要使用 `DataInputStream` 类 (`FilterInputStream` 的子类)。

`DataInput` 的接口由 `DataInputStream` 类和 `RandomAccessFile` 类 (下面将要讨论) 完成, 它们都需要从一个流中读取原始数据类型。`DataInputStream` 能使程序从一个 `InputStream` 中读取二进制数据。我们采用的典型方法是将 `DataInputStream` 和 `FileInputStream` 连接起来。`DataInput` 的接口包括 `read` (对于字节)、`readBoolean`、`readByte`、`readChar`、`readDouble`、`readFloat`、`readFully` (对于字节数组)、`readInt`、`readLine`、`readLong`、`readShort`、`readUnsignedByte`、`readUnsignedShort`、`readUTF` (对标准格式字符串) 和 `skipBytes` 这些方法。

`DataOutput` 的接口由 `DataOutputStream` (`FilterOutputStream` 的子类) 和 `RandomAccessStream` 类完成, 它们都需要向一个 `OutputStream` 中写入原始数据类型。`DataOutputStream` 能使程序向一个

OutputStream中写入二进制数据。我们采用的典型方法是先将DataOutputStream和FilterOutputStream连接起来。DataInput的接口包括: flush、size、write(对于字节)、write(对于字节数组)、writeBoolean、writeByte、writeBytes、writeChar、writeChars(对于标准字符串)、writeDouble、writeFloat、writeInt、writeLong、writeShort和writeUTF这些方法。

设置缓冲是一种I/O操作的增强技术。通过一个BufferedOutputStream(FilterOutputStream的子类), 每条输出指令都不会导致数据向输出设备进行实际的物理转换, 而是每个输出操作都直接面向称为缓冲区的一块内存区域。这块区域足够大, 能够放下许多输出操作的数据。然后每次当占满缓冲区以后, 一个较大的物理输出操作就会完成对输出设备的实际输出。这种直接面向内存中的输出缓冲区的操作通常称为逻辑输出操作。

#### 性能提示 15.1

因为通常的物理输出操作与处理器速度相比非常慢, 所以在正常情况下, 利用缓冲区输出会比非缓冲区输出在性能上有相当大的提高。

通过一个BufferedInputStream类(FilterInputStream的子类), 许多逻辑数据块将由一个大的物理输入操作从文件中读入内存缓冲区。当程序需要新的数据块时, 就可以从缓冲区中取出数据(有时也称为逻辑输入操作)。当缓冲区为空时, 输入设备的下一个实际的物理输入操作会读入下一组“逻辑”数据块。这样, 实际的物理输入操作的次数与程序发出的读请求的次数相比就会少多了。

#### 性能提示 15.2

因为通常的物理输入操作与通常的处理器速度相比非常慢, 所以在正常情况下, 利用缓冲区输入会比非缓冲区输入在性能上有相当大的提高。

利用BufferOutputStream类, 能够在任何时候强迫一个未充满的缓冲区向设备输出, 同时也会执行下面的外部清除(flush方法):

```
testBufferedOutputStream.flush();
```

PushBackInputStream类(FilterInputStream的一个子类)用来完成超出大部分用户需要的应用程序。本质上, 这种读取一个PushBackInputStream的应用程序从流中读入若干字节, 并形成由几个字节组成的集合。有时, 为了判断一个数据收集是否结束, 应用程序不得不读取前一个集合“结束之后”的第一个字符。一旦程序断定当前的数据收集已经结束, 就会将多读的字符送回到流中。许多程序使用PushBackInputStream, 例如编译器就利用其来解析输入, 也就是把输入分成有意义的单元(例如Java编译器必须识别的关键字、标识符和运算符)。

如果我们想把一个来自于文本输入文件的文本输入流分解成称为标记(token)的有意义的单元, 那么StreamTokenizer类(Object的一个子类)是很有用的。StreamTokenizer类的作用同我们在第8章中讨论的StringTokenizer类是相似的。

对于直接访问的应用程序, 例如航班预订系统和销售系统这样的交易处理应用程序, RandomAccessFile类(Object的一个子类)是很有用的。对于顺序访问文件, 每一个连续的输入/输出请求都将读取或写入文件中下一个连续的数据集合。而对于随机访问文件, 每一个连续的输入/输出请求都会指向文件的任意部分, 这可能与文件中上一次请求所引用的部分相距很远。直接访问应用程序提供了快速访问大文件中特定数据的功能; 当人们正在等待回答并且这些答案很快就要用到的时候, 当人们变得不耐烦并且要“换个地方做生意”的时候, 就会经常使用这样的应用程序。

Java的I/O流包括从内存中的字节数组输入数据和向内存中的字节数组输出数据的功能。利用ByteArrayInputStream类(InputStream抽象类的子类), 可以从内存中的字节数组输入; 而利用

`ByteArrayOutputStream` 类 (`OutputStream` 抽象类的子类), 可以向内存中的字节数组输出。一个字节数组的 I/O 应用程序完成对数据的确认工作。程序每次从输入流向字节数组输入完整的一行, 然后确认例程仔细检查了字节数组的内容, 并且在需要的时候更正数据。在确定输入数据的格式正确之后, 程序才能继续从字节数组输入。向字节数组输出利用了强大的 Java 流的格式化输出功能。为了模拟编辑过的屏幕格式, 需要将数据整理成字符数组的格式。而为了保存屏幕图像, 则应将该数组写入磁盘文件。

`StringBufferInputStream` 类 (`InputStream` 抽象类的子类) 从 `StringBuffer` 对象中得到输入的信息。

`SequenceInputStream` 类 (`InputStream` 抽象类的子类) 能够连接几个 `InputStream`, 以便程序可以把这些流看成是一个连续的 `InputStream`。在每个输入流结束的时候, 关闭这个输入流, 继而打开序列中的下一个流。

`LineNumberInputStream` 类 (`FilterInputStream` 类的子类) 总能知道正在读取的文件的行号。

多数 Java 用户一般不使用 `FileDescriptor` 类的对象, 因此我们省略对 `FileDescriptor` 类的讨论。

`File` 类能够使程序获得一个文件或目录的信息。我们将在 15.12 节进一步讨论 `File` 类。

## 15.4 创建顺序访问文件

Java 文件中没有结构。这样一来, 诸如“记录”这样的概念在 Java 文件中就不存在了。因此, 程序员必须组织文件以符合应用程序的需求。在下面这个例子中, 我们将看到程序员怎样将一个简单的记录结构加入一个文件中。我们首先介绍程序, 然后再详细地分析它。

图 15.4 的程序创建了一个简单的顺序访问文件, 它可以在应收账款系统中使用, 以便帮助一个公司管理有信用的客户所应支付的钱款。该程序获得每个客户的账号、该客户的姓名和客户的欠额(即客户所欠公司的货物和服务费用的总和)。每个客户的这些数据组成了该客户的一个记录。在这个应用程序中, 账号作为记录的关键字; 也就是文件将按照账号的顺序进行创建和维护。该程序假定用户按照账号顺序输入记录, 一个性能良好的收入账目系统会提供排序功能, 这样用户就能以任何顺序输入记录, 这些记录在排序后将写入文件。

```

1      // Fig. 15.4: CreateSeqFile.java
2      // This program uses TextFields to get information from the
3      // user at the keyboard and writes the information to a
4      // sequential file.
5      import java.io.*;
6      import java.awt.*;
7
8      public class CreateSeqFile extends Frame {
9
10         // Application window components
11         TextField account, // where user enters account number
12             fName,        // where user enters first name
13             lName,        // where user enters last name
14             balance;       // where user enters balance
15         Button enter,      // send record to file
16             done;         // quit program
17         Label acctLabel,   // account label
18             fNameLabel,   // first name label
19             lNameLabel,   // last name label
20             balLabel;     // balance label

```

```
21
22     // Application other pieces
23     DataOutputStream output; // enables output of data to file
24
25     // Constructor -- initialize the Frame
26     public CreateSeqFile()
27     {
28         super( "Create Client File" );
29     }
30
31     public void addRecord()
32     {
33         int acct = 0;
34         Double d;
35
36         acct = (new Integer( account.getText() )).intValue();
37
38         // output the values to the file
39         try {
40             if ( acct > 0 ) {
41                 output.writeInt( acct );
42                 output.writeUTF( fName.getText() );
43                 output.writeUTF( lName.getText() );
44                 d = new Double ( balance.getText() );
45                 output.writeDouble( d.doubleValue() );
46             }
47         }
48         catch ( IOException e ) {
49             System.err.println( "Error during write to file\n" +
50                                 e.toString() );
51             System.exit( 1 );
52         }
53
54         // clear the TextFields
55         account.setText( "" );
56         fName.setText( "" );
57         lName.setText( "" );
58         balance.setText( "" );
59     }
60
61     // Setup the window for the application with
62     // TextFields and Buttons
63     public void setup()
64     {
65         resize( 300, 150 );
66         setLayout( new GridLayout( 5, 2 ) );
67
68         // create the components of the Frame
69         account = new TextField( 20 );
70         acctLabel = new Label( "Account Number" );
71         fName = new TextField( 20 );
72         fNameLabel = new Label( "First Name" );
73         lName = new TextField( 20 );
74         lNameLabel = new Label( "Last Name" );
75         balance = new TextField( 20 );
76         balLabel = new Label( "Balance" );
```

```

77     enter = new Button( "Enter" );
78     done = new Button( "Done" );
79
80     // add the components to the frame
81     add( acctLabel ); // add label
82     add( account ); // add TextField
83     add( fNameLabel ); // add label
84     add( fName ); // add TextField
85     add( lNameLabel ); // add label
86     add( lName ); // add TextField
87     add( balLabel ); // add label
88     add( balance ); // add TextField
89     add( enter ); // add button
90     add( done ); // add button
91
92     show(); // show the frame
93
94     // Open the file
95     try {
96         output = new DataOutputStream(
97             new FileOutputStream( "client.dat" ) );
98     }
99     catch ( IOException e ) {
100         System.err.println( "File not opened properly\n" +
101             e.toString() );
102         System.exit( 1 );
103     }
104 }
105
106 // Cleanup--add the current information (if valid),
107 // flush remaining output to file and close file.
108 public void cleanup()
109 {
110     if ( !account.getText().equals("") )
111         addRecord();
112
113     try {
114         output.flush();
115         output.close();
116     }
117     catch ( IOException e ) {
118         System.err.println( "File not closed properly\n" +
119             e.toString() );
120         System.exit( 1 );
121     }
122 }
123
124 // Process the event when the user closes the window
125 // or presses the "Done" button
126 public boolean handleEvent( Event event )
127 {
128     // User closed the window or clicked the done button--
129     // be sure info is written to file and exit program
130     if ( event.id == Event.WINDOW_DESTROY ||
131         event.target == done ) {
132         cleanup(); // write data, close file, etc.

```

```
133
134         hide();
135         dispose(); // release system resources
136         System.exit( 0 );
137         return true;
138     }
139
140     // User clicked Enter button to send record to file
141     if ( event.target instanceof Button ) {
142         if ( event.arg.equals( "Enter" ) ) {
143             addRecord();
144             return true;
145         }
146     }
147
148     return super.handleEvent( event );
149 }
150
151 // Instantiate a CreateSeqFile object and start the program
152 public static void main( String args[ ] )
153 {
154     CreateSeqFile accounts = new CreateSeqFile();
155
156     accounts.setup();
157 }
158 }
```

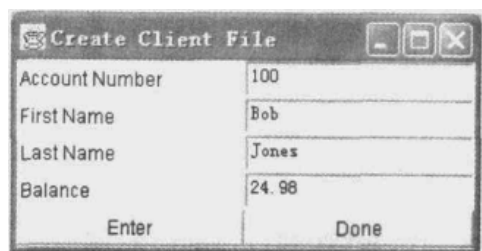


图 15.4 创建一个顺序访问文件

现在让我们来解释这个程序。正如前面所述，文件是通过创建流类 `FileInputStream` 和 `FileOutputStream` 的对象而打开的。在图 15.4 中，文件为输出而打开，因此创建了 `FileOutputStream` 的一个对象，并将参数——文件名传递给该对象的构造函数。为输出而打开的现有文件将被截尾，也就是将丢弃文件中所有的数据。如果这个指定的文件不存在，那么就利用给定的文件名创建一个。

在这个程序中，我们利用 `DataOutputStream` 输出数据，它通过所谓的流对象链接技术来与 `FileOutputStream` 相联系。例如，当创建 `DataOutputStream` 的对象 `output` 时（如图 15.4 中的第 96 行和第 97 行所示），`FileOutputStream` 的对象将作为参数而提供给它的构造函数：

```
output = new DataOutputStream(
    new FileOutputStream( "client.dat" ));
```

上面的语句创建了名为 `output` 的 `DataOutputStream` 的对象，它与文件 `client.dat` 相关联。参数 `"client.dat"` 将传递给要打开该文件的 `FileOutputStream` 的构造函数。这样就同该文件建立了“一条通信线路”。



### 常见编程错误 15.1

用户为输出打开了一个现有文件，并且用户的本意是想保存该文件；但是，文件在没有给出警告的情况下就将丢弃了。

### 常见编程错误 15.2

使用一个不正确的 `FileOutputStream` 的对象来与文件相关联。

### 常见编程错误 15.3

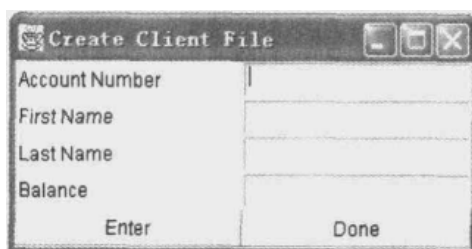
程序中，在没有打开一个文件的前提下试图引用该文件。

当创建一个 `FileOutputStream` 的对象并试图打开一个文件时，程序需要测试该打开操作是否成功。如果操作失败，将会抛出一个 `IOException` 异常，程序必须捕获该异常，如下所示（图 15.4 的第 95 行 ~ 第 104 行）：

```
try{
    output = new DataOutputStream(new FileOutputStream("client.dat"));
}
catch ( IOException e ) {
    System.err.println( "File not opened properly \n" + e. toString() );
    System.exit( 1 );
}
```

如果由于打开操作不成功而抛出了 `IOException` 异常，则输出错误信息 “File not opened properly”，后面紧跟着异常对象中的错误信息，然后调用 `System` 类的 `exit` 方法来结束程序。`exit` 的参数返回给引发该程序的环境。参数 0 表示程序正常终止，其他数值表示程序由于某种错误而终止。调用环境（很可能是操作系统）利用 `exit` 的返回值来正确响应错误。在打开文件时抛出 `IOException` 异常的一些可能原因是：为了读数据而试图打开一个不存在的文件，在没有读权限的情况下试图打开一个文件，在没有足够磁盘空间的情况下打开一个文件并写入数据。

如果文件成功打开，则程序就准备对其中的数据进行操作。下面是允许用户输入数据的窗口：



假定程序按照正确的记录号顺序输入了数据。当输入完一个记录后，用户按下 `Enter` 按钮以便向文件写入该记录，这样就引发了我们的 `CreateSeqFile` 类的 `addRecord` 方法来执行写操作。在这个方法中，通过调用 `DataOutputStream` 类中的 `writeInt`（写一个整数值）、`writeUTF`（写一个字符串）和 `writeDouble`（写一个双精度浮点数）方法，将记录的每个域分别写入文件。`addRecord` 方法的第 41 行 ~ 第 46 行接收用户输入的每个数据项并立即将它们写入文件。用户按下 `Done` 按钮来告诉程序没有数据需要输入了，具有读文件功能的程序（参见 15.5 节）将检索这些数据。

一旦用户按下 `Done` 按钮，程序就会终止。然而，在终止前程序会判断是否有要写入磁盘的记录，同时刷新输出流，以确定所有的数据都已送往文件并关闭输出流。程序员可以使用如下所示的 `close` 方法从外部关闭 `FileOutputStream` 的对象：

```
output.close();
```

或者在进行无用单元回收处理时，从内部关闭该文件。

#### 性能提示 15.3

总是在外部释放资源，并尽可能早地断定不再需要该资源，这样可以使读者的程序或其他程序重新使用该资源，从而提高资源的利用率。

当使用链接的流对象时，应使用最外面的对象（在本例中是 `DataOutputStream`）来关闭文件。

#### 性能提示 15.4

一旦知道程序将不再引用某一文件时，就从外部关闭该文件。对于一个需要长时间连续运转的程序，在得知不再引用某一文件之后，关闭该文件会减少资源的使用。这种操作也会使程序更加清晰。

在图 15.4 所示的程序中，我们输入 5 个账号信息（如图 15.5 所示），然后按下 Done 按钮来说明数据输入已经完成。该程序不显示数据记录在文件中的实际情况。为了验证该文件已经成功创建，我们将在下一节编写一个程序来读取文件。

实例数据			
100	Bob	Jones	24.98
200	Steve	Doe	345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16
500	Sue	Rich	224.62

图 15.5 图 15.4 所示程序的实例数据

## 15.5 从顺序访问文件中读取数据

我们将数据存储在文件中，以便在需要对它们进行操作时能将其检索出来。上一节讲述了如何创建一个顺序访问文件，这一节我们讨论怎样从一个文件中顺序地读取数据。

图 15.6 所示的程序从 `client.dat` 文件（由图 15.4 的程序创建）中读取记录并打印这些记录的内容。通过创建一个 `FileInputStream` 的对象，可以打开文件以便输入，并将要打开的文件名作为一个参数传递给 `FileInputStream` 的构造函数。在图 15.4 中，我们利用 `DataOutputStream` 的一个对象向文件 `client.dat` 中写入数据。数据必须按照它们写入文件时的同样格式从文件中读取出来。因此，在这个程序中，我们使用同 `FileInputStream` 链接的 `DataInputStream` 来读取数据，如图 15.6 第 66 行和第 67 行所示：

```
input = new DataInputStream(  
    new FileInputStream("client.dat"));
```

上面的语句创建了一个与文件 `client.dat` 相联系的、名为 `input` 的 `DataInputStream` 的对象。在打开文件时，参数“`client.dat`”将传递给 `FileInputStream` 的构造函数。这样，程序就同该文件建立了一条“通信路径”。

```
1 // Fig. 15.6: ReadSeqFile.java  
2 // This program reads a file sequentially and displays each  
3 // record in text fields.
```

```
4    import java.io.*;
5    import java.awt.*;
6
7    public class ReadSeqFile extends Frame {
8
9        // Application window components
10       TextField acct,    // displays account number
11               fName,    // displays first name
12               lName,    // displays last name
13               bal;      // displays balance
14       Button next,      // display next record
15               done;     // quit program
16       Label acctLabel, // account label
17               fNameLabel, // first name label
18               lNameLabel, // last name label
19               balLabel;  // balance label
20
21       // Application other pieces
22       DataInputStream input; // file from which data is read
23       boolean moreRecords = true;
24
25       // Constructor -- initialize the Frame
26       public ReadSeqFile()
27       {
28           super( "Read Client File" );
29       }
30
31       public void readRecord()
32       {
33           int account;
34           String first, last;
35           double balance;
36
37           // input the values from the file
38           try {
39               account = input.readInt();
40               first = input.readUTF();
41               last = input.readUTF();
42               balance = input.readDouble();
43
44               acct.setText( String.valueOf( account ) );
45               fName.setText( String.valueOf( first ) );
46               lName.setText( String.valueOf( last ) );
47               bal.setText( String.valueOf( balance ) );
48           }
49           catch ( EOFException eof ) {
50               moreRecords = false;
51           }
52           catch ( IOException e ) {
53               System.err.println( "Error during read from file\n" +
54                                   e.toString() );
55               System.exit( 1 );
56           }
57       }
58
59       // Setup the window for the application with TextFields
```

```
60      // and Buttons
61      public void setup()
62      {
63
64          // Open the file
65          try {
66              input = new DataInputStream(
67                  new FileInputStream( "client.dat" ) );
68          }
69          catch ( IOException e ) {
70              System.err.println( "File not opened properly\n" +
71                                  e.toString() );
72              System.exit( 1 );
73          }
74
75          resize( 300, 150 );
76          setLayout( new GridLayout( 5, 2 ) );
77
78          // create the components of the Frame
79          acct = new TextField( 20 );
80          acctLabel = new Label( "Account Number" );
81          fName = new TextField( 20 );
82          fNameLabel = new Label( "First Name" );
83          lName = new TextField( 20 );
84          lNameLabel = new Label( "Last Name" );
85          bal = new TextField( 20 );
86          balLabel = new Label( "Balance" );
87          next = new Button( "Next" );
88          done = new Button( "Done" );
89
90          // add the components to the Frame
91          add( acctLabel );    // add label
92          add( acct );         // add TextField
93          add( fNameLabel );   // add label
94          add( fName );        // add TextField
95          add( lNameLabel );   // add label
96          add( lName );        // add TextField
97          add( balLabel );     // add label
98          add( bal );          // add TextField
99          add( next );         // add button
100         add( done );          // add button
101
102         show();               // show the Frame
103     }
104
105     // Cleanup--close the input file.
106     public void cleanup()
107     {
108         try {
109             input.close();
110         }
111         catch ( IOException e ) {
112             System.err.println( "File not closed properly\n" +
113                                 e.toString() );
114             System.exit( 1 );
115         }
116     }
```

```

116     }
117
118     public boolean action( Event event, Object o )
119     {
120         if ( event.target instanceof Button )
121             if ( event.arg.equals( "Next" ) )
122                 readRecord();
123
124         return true;
125     }
126
127     // Process the event when the user closes the window
128     // or presses the "Done" button
129     public boolean handleEvent( Event event )
130     {
131         // End of file reached, user closed window, or
132         // user clicked the Done button.
133         if ( moreRecords == false ||
134             event.id == Event.WINDOW_DESTROY ||
135             event.target == done ) {
136             cleanup(); // close file
137
138             hide();
139             dispose(); // release system resources
140             System.exit( 0 );
141             return true;
142         }
143
144         // May need to process event still
145         return super.handleEvent( event );
146     }
147
148     // Instantiate a ReadSeqFile object and start the program
149     public static void main( String args[ ] )
150     {
151         ReadSeqFile accounts = new ReadSeqFile();
152
153         accounts.setup();
154     }
155 }

```

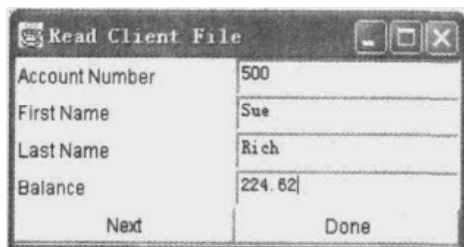


图 15.6 读取一个顺序文件

在创建一个 `FileInputStream` 的对象并试图打开一个文件的时候, 程序要测试打开操作是否成功。如果操作失败, 将会抛出一个 `IOException` 异常, 程序必须捕获该异常, 如下所示如 (图 15.6 的第 65 行 ~ 第 73 行):

```
try{
    input = new DataInputStream(
        new FileInputStream("client.out"));
}
catch ( IOException e ) {
    System.err.println( "File not opened properly \n" +
        e.toString() );
    System.exit( 1 );
}
```

如果由于打开操作不成功而抛出了IOException异常,则会输出错误信息“File not opened properly”,后面紧跟着异常对象中的错误信息,然后调用System类的exit方法来结束程序。如果成功打开了文件,那么程序将准备对数据进行操作。

每次当用户按下“Next”按钮时,程序都将读取一条记录。这个动作将抛出ReadSeqFile类的readRecord方法。下面几行:

```
account = input.readInt();
first = input.readUTF();
last = input.readUTF();
balance = input.readDouble();
```

从文件中读取一条记录的信息。如果读到文件结束符,就抛出EOFException异常,它将moreRecord标志设置为false。如果此时用户试图通过按下Next按钮来读下一条记录,则程序将会终止。

前面的代码在首次执行后,account的值是100,first的值是“Bob”,last的值是“Jones”,balance的值是24.98。以后在每次执行前面的代码时,都会从文件中把下一条记录读入account、first、last和balance。下面几行程序把记录显示在文本字段中:

```
acct.setText( String.valueOf( account ) );
fName.setText( String.valueOf( first ) );
lName.setText( String.valueOf( last ) );
bal.setText( String.valueOf( balance ) );
```

如果用户按下Done按钮(或者在moreRecords的值为false时按下Next按钮),就会引发cleanUp方法关闭该文件,该方法调用了DataInputStream的对象input的关闭方法。程序通过调用System类的exit方法来终止执行。

为了从文件中顺序地检索数据,程序从文件的起始位置开始读取数据,并且一直连续地读完所有数据,直到找到所需要的数据为止。在程序的执行过程中,可能需要顺序地访问文件若干次(从文件的起始位置)。目前,在Java编程语言中,对于下一个输入或输出操作,还没有一个方法能够在FileInputStream和FileOutputStream中重定位文件位置指针(进行读或写操作的下一个字节的字节号)。因此,FileInputStream的对象只能通过关闭一个文件并重新打开它,才能将文件位置指针重新定位于文件的起始位置。

#### 性能提示 15.5

为了将文件位置指针重新定位到文件的起始位置,需要关闭该文件并重新打开它,这样的操作对计算机来说是要耗费时间的;如果频繁地进行此类操作,程序的运行速度将会降低。

图15.7所示的程序为一个信贷管理者分别显示余额为0(即不欠公司任何钱款的顾客)、有贷款余额(即公司的债权人)以及有借款余额(即过去接受货物和服务而欠有公司钱款的顾客)的顾客的账目信息。

```

1 // Fig. 15.7: CreditInquiry.java
2 // This program reads a file sequentially and displays the
3 // contents in a text area based on the type of account the
4 // user requests (credit balance, debit balance or zero balance.
5 import java.io.*;
6 import java.awt.*;
7
8 public class CreditInquiry extends Frame {
9     // Application window components
10    TextArea recordDisplay; // where records are displayed
11    Button done;             // quit program
12    Button credit,           // account type to display
13           debit,            // account type to display
14           zero;             // account type to display
15    Panel buttonPanel;      // panel for buttons
16    String accountType;     // Account type to display
17
18    // Application other pieces
19    File file;
20    DataInputStream input; // file from which data is read
21
22    // Constructor -- initialize the Frame
23    public CreditInquiry()
24    {
25        super( "Credit Inquiry Program" );
26    }
27
28    public void readRecords()
29    {
30        int acct;
31        String first, last;
32        double bal;
33
34        // input the values from the file
35        try {
36            recordDisplay.setText( accountType + ":\n" );
37
38            while ( true ) {
39                acct = input.readInt();
40                first = input.readUTF();
41                last = input.readUTF();
42                bal = input.readDouble();
43
44                if ( shouldDisplay( bal ) )
45                    recordDisplay.appendText( acct +
46  "\t" + first + "\t" + last +
47  "\t" + bal + "\n" );
48            }
49        }
50        catch ( EOFException eof ) {
51            // do nothing on EOF exception
52        }
53        catch ( IOException e ) {
54            System.err.println( "Error during read from file\n" +
55                                e.toString() );

```

```
56         System.exit( 1 );
57     }
58 }
59
60 public boolean shouldDisplay( double balance )
61 {
62     if ( accountType.equals( "Credit balances" ) &&
63         balance < 0 )
64         return true;
65
66     if ( accountType.equals( "Debit balances" ) &&
67         balance > 0 )
68         return true;
69
70     if ( accountType.equals( "Zero balances" ) &&
71         balance == 0 )
72         return true;
73
74     return false;
75 }
76
77 // Setup the window for the application with TextArea
78 // and Buttons
79 public void setup()
80 {
81     resize( 400, 150 );
82
83     // create the components of the Frame
84     recordDisplay = new TextArea( 4, 40 );
85     buttonPanel = new Panel();
86     done = new Button( "Done" );
87     credit = new Button( "Credit balances" );
88     debit = new Button( "Debit balances" );
89     zero = new Button( "Zero balances" );
90     buttonPanel.add( credit );
91     buttonPanel.add( debit );
92     buttonPanel.add( zero );
93     buttonPanel.add( done );
94
95     // add the components to the Frame
96     add( "Center", recordDisplay );    // add TextArea
97     add( "South", buttonPanel );      // add Button
98
99     show();                          // show the Frame
100 }
101
102 // Process the event when the user closes the window
103 // or presses the "Done" button
104 public boolean handleEvent( Event event )
105 {
106     // User closed the window or clicked the Done button
107     // Clean up window and exit program
108     if ( event.id == Event.WINDOW_DESTROY ||
```



```
109         event.target == done ) {
110             hide();
111             dispose(); // release system resources
112             System.exit( 0 );
113             return true;
114         }
115
116         // User selected type of account to display
117         if ( event.target instanceof Button ) {
118             accountType = event.arg.toString();
119
120             // Open the file
121             try {
122                 input = new DataInputStream(
123                     new FileInputStream( "client.dat" ) );
124             }
125             catch ( IOException e ) {
126                 System.err.println( e.toString() );
127                 System.exit( 1 );
128             }
129
130             readRecords();
131
132             // Close the file
133             try {
134                 input.close();
135             }
136             catch ( IOException e ) {
137                 System.err.println( "File not closed properly\n" +
138                                     e.toString() );
139                 System.exit( 1 );
140             }
141         }
142
143         return super.handleEvent( event );
144     }
145
146     // Instantiate a CreditInquiry object and start the program
147     public static void main( String args[ ] )
148     {
149         CreditInquiry accounts = new CreditInquiry();
150
151         accounts.setup();
152         accounts.show();
153     }
154 }
```

图 15.7 信贷调查程序

该程序显示了若干个按钮，它们允许信贷管理者获得信贷信息。“Credit balances”按钮会生成一个贷款余额的账目列表，“Debit balances”按钮会生成一个借款余额的账目列表，“Zero balances”按钮会生成一个余额为0的账目列表，而 Done 按钮将终止程序执行。输出示例如图 15.8 所示。

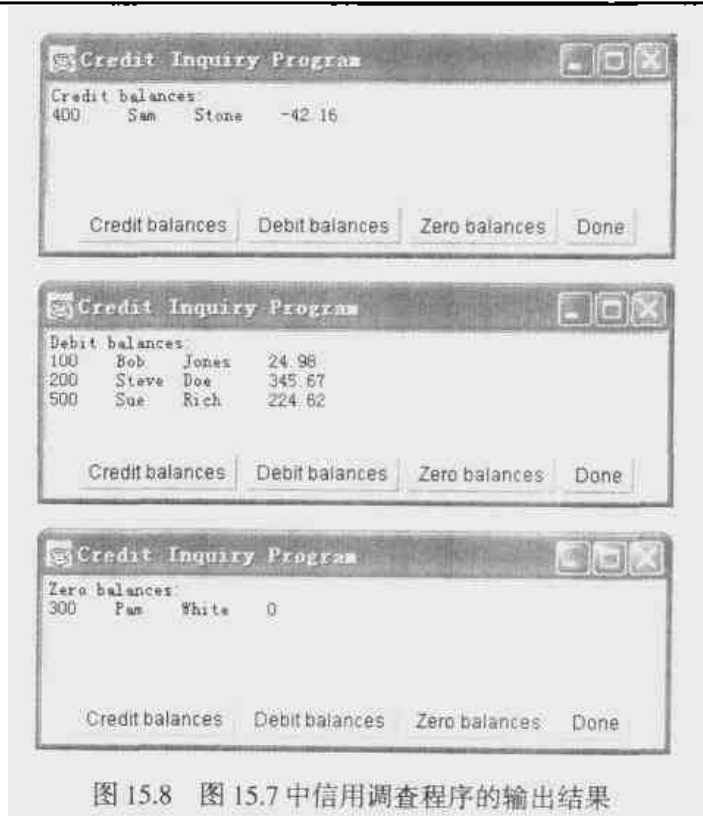


图 15.8 图 15.7 中信用调查程序的输出结果

指定类型的记录将显示在称为 `recordDisplay` 的 `TextArea` 中。信贷管理者通过按下某个按钮来选择账目类型，程序读取整个文件，并判断每条记录是否符合用户所选的类型，这样记录信息便被收集起来。按下 `Done` 按钮之外的其他按钮会使 `handleEvent` 方法将 `accountType` 设置成合适的类型，打开 `client.dat` 文件并调用 `readRecords` 方法。`readRecords` 方法循环读取文件中的每一条记录。调用 `shouldDisplay` 方法来判断当前记录是否满足所要求的类型。如果该方法返回 `true`，则当前记录的账目信息就将添加到称为 `records` 的 `StringBuffer` 中。当读到文件结束符时，就为 `recordDisplay` 调用 `setText` 方法，并把 `record.toString()` 的结果作为字符串显示出来。

## 15.6 更新顺序访问文件

15.4 节中描述的经过格式化并写入顺序访问文件中的数据，在没有读取和写入文件中的所有数据前是不能修改的。例如，如果需要将名字 `White` 修改为 `Worthington`，我们就不能简单地通过重写原来的名字来完成该要求。

可以实现下面这样的更新，但是方法不是很简单。例如，为了改变以前的名字，顺序文件中 `White` 以前的记录都要复制到一个新文件中，然后将更新后的记录写入新文件中，最后把 `White` 之后的所有记录复制到新文件中。为了更新一个记录，我们需要对文件中所有的记录进行操作。只有在一次需要更新一个文件中的许多记录时，这种技术才可以接受。

## 15.7 随机访问文件

至今为止，我们已经知道如何创建顺序访问文件，以及如何搜索它们并得到指定的信息。顺序访问文件对于称为“即时访问”的应用程序是不合适的，因为后者需要立即定位存放某一信息的记

录。一些流行的即时访问的应用程序有航班预订系统、银行系统、销售点系统、自动柜员机和其他各种需要迅速访问特定数据的交易处理系统。银行可能有成百上千甚至数百万个顾客；然而在使用自动柜员机时，它可以在几秒钟内查出特定的账号内是否有足够的存款。如果使用随机访问文件，这种即时访问是可能实现的。在不用搜索其他记录的前提下，可以直接并迅速读取随机访问文件中的某个记录。

正如我们已经提到过的，Java不能将结构强加于一个文件之上。因此，如果想要使用随机访问文件的应用程序，就必须自己创建它们。我们能够使用各种不同的技术来创建随机访问文件，其中最简单的方法可能就是要求一个文件中的所有记录都具有相同的固定长度。

为了计算（利用记录大小和记录关键字）任何记录相对于文件起始位置的准确位置，我们可以在程序中使用定长记录，这样能使计算变得容易些。不久我们就会看到，这样做能使立即访问某个特定记录的操作变得容易，即使在大文件中也不例外。

图 15.9 给出了 Java 的一个随机访问文件的例子，这个文件由若干固定长度的记录组成（每个记录的长度为 100 字节）。一个随机访问文件就像一列有许多车厢的火车——有些车厢是空的，而有些则装了东西。

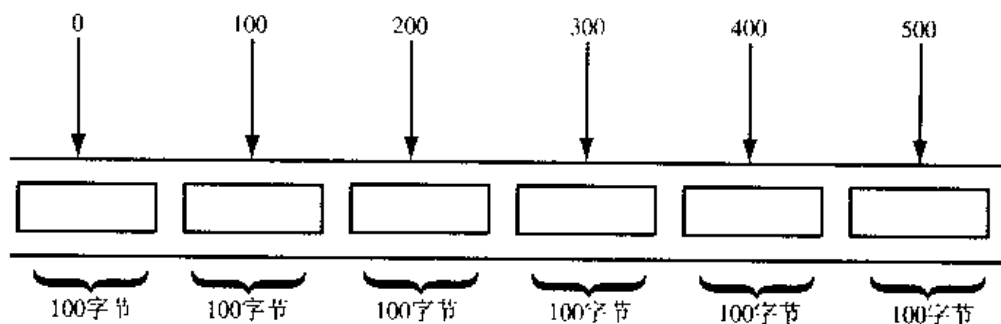


图 15.9 Java 中的一个随机访问文件

我们可以不破坏文件中的其他数据而将数据插入一个随机访问文件，也可以不重写整个文件而更新或删除以前存储的数据。在以后几节中，我们将解释如何创建一个随机访问文件，输入数据，顺序地和随机地读取数据，更新数据，以及删除不再需要的数据。

## 15.8 创建随机访问文件

`RandomAccessFile` 类的对象具有我们前面几节所讨论的 `DataInputStream` 和 `DataOutputStream` 类对象所具有的全部功能。当一个 `RandomAccessFile` 流同一个文件相关联时，数据就从文件位置指针所确定的文件位置开始读或写，同时所有的数据都将作为原始数据类型而对其进行读或写。如果写入一个整数值，就会向文件输出 4 个字节；如果读取一个双精度浮点数，就会从文件中输入 8 个字节。我们能够保证识别所有不同的数据类型，因为 Java 中所有的原始数据类型都有固定大小，而不管其计算平台是什么。

随机访问文件处理程序很少向文件中写入一个单独的域。正常情况下，它们一次写入一个对象，如同下面的例子所给出的。

考虑下面的问题说明：

创建一个交易处理程序，它能存储一个公司中多达 100 个固定长度的记录。每条记录的组成部

分应有账号（作为记录关键字）、姓氏、名称和余额。程序应能够更新一个账户、插入一个账户和删除一个账户。

下面几节将介绍创建这个信用管理系统所需要的技术。图15.10包括下面4个程序使用的Record类，这4个程序从“credit.dat”文件中读取和写入记录。

Record类包含4个描述记录中的信息内容的实例变量——account、lastName、firstName和balance，以及三个对记录进行操作的方法。write方法从RandomAccessFile的对象中输出一条记录信息，该对象是作为一个参数传递给方法write的。程序中又使用writeInt方法来输出账号，使用write方法来输出以字节数组形式存放的姓氏和名字（每个含有15个字节），以及使用writeDouble方法来输出余额。

```
1      // Fig. 15.10: Record.java
2      // Record class for the RandomAccessFile programs.
3      import java.io.*;
4
5      public class Record {
6          int account;
7          String lastName;
8          String firstName;
9          double balance;
10
11         // Read a record from the specified RandomAccessFile
12         public void read( RandomAccessFile file ) throws IOException
13         {
14             account = file.readInt();
15             byte b1[] = new byte[ 15 ];
16             file.readFully( b1 );
17             firstName = new String( b1, 0 );
18             byte b2[] = new byte[ 15 ];
19             file.readFully( b2 );
20             lastName = new String( b2, 0 );
21             balance = file.readDouble();
22         }
23
24         // Write a record to the specified RandomAccessFile
25         public void write( RandomAccessFile file ) throws IOException
26         {
27             file.writeInt( account );
28
29             byte b1[] = new byte[ 15 ];
30
31             if ( firstName != null )
32                 firstName.getBytes( 0, firstName.length(), b1, 0 );
33
34             file.write( b1 );
35
36             byte b2[] = new byte[ 15 ];
37
38             if ( lastName != null )
39                 lastName.getBytes( 0, lastName.length(), b2, 0 );
40
41             file.write( b2 );
42             file.writeDouble( balance );
```

```

43     }
44
45     // NOTE: This method contains a hard coded value for the
46     // size of a record of information.
47     public int size() { return 42; }
48 }

```

图 15.10 随机访问文件中使用的 Record 类

read 方法从 RandomAccessFile 的对象中输入一条记录信息, 该对象是作为一个参数传递给 read 方法的。RandomAccessFile 的 readInt 方法和 readDouble 方法分别用于输入账号和余额。readFully 方法用来将姓氏和名字读入 15 字节长的数组。这些数组作为 String 对象的初值将分别赋给 firstName 和 lastName 对象。size 方法返回按字节数计算的记录的大小。该方法定义为从外部返回值 42, 因为每条记录占用 42 个字节大小的磁盘空间 (4 个字节分配给 int account, 15 个字节分配给转化为字节数组的 firstName, 15 个字节分配给转化为字节数组的 lastName, 8 个字节分配给 double balance)。

图 15.11 给出了打开一个随机访问文件并向磁盘写入数据的例子。这个程序使用了我们 Record 类中的 write 方法, 将 “credit.dat” 文件中的 100 条记录都初始化为空对象。每个空对象中 account 为 0, firstName 字符串是空的 (由空的引号标志表示), lastName 字符串是空的, 并且 balance 为 0.0。初始化文件的目的是为了创建存放账户数据所用的空间, 同时使我们能够使用随后的程序来判断每条记录是空的还是存有数据。

```

1  //Fig. 15.11: CreateRandFile.java
2  //This program creates a random access file sequentially
3  //by writing 100 empty records to disk.
4  import java.io.* ;
5
6  public class CreateRandFile {
7      private Record blank;
8      RandomAccessFile file;
9
10     public CreateRandFile()
11     {
12         blank = new Record();
13
14         try{
15             file = new RandomAccessFile("credit.dat", "rw");
16         }
17         catch( IOException e){
18             System.err.println("File Not opened properly\n" +
19                               e.toString() );
20             System.exit(1);
21         }
22     }
23
24     public void create()
25     {
26         try {
27             for (int i = 0; i < 100; i ++){
28                 blank.write(file);
29             }
30         } catch ( IOException e) {
31             System.err.println(e.toString() );

```

```

32         }
33     }
34
35     public static void main( String atgs[] )
36     {
37         CreateRandFile accounts = new CreateRandFile();
38
39         accounts.create();
40     }
41 }

```

图 15.11 创建一个随机访问文件

在图 15.11 中，下面的程序行（第 14 行~第 21 行）：

```

try {
    file = new RandomAccessFile( "credit.dat", "rw" );
}
catch (IOException e) {
    System.err.println( " File not opened properly\n" +
                        e. toString() );
    System.exit( 1 );
}

```

试图在这个程序中打开文件“credit.dat”。将两个参数——文件名和文件打开方式传递给 RandomAccessFile 的构造函数。一个 RandomAccessFile 的文件打开方式可以为“r”（为了读而打开文件）或“rw”（为了读和写而打开文件）。如果在打开文件的过程中抛出了 IOException 异常，就会终止程序。如果正确打开了文件，程序就使用一个 for 结构来将下面的语句执行 100 次：

```
blank.write( file );
```

这条语句将对象 blank 写入与 RandomAccessFile 的对象 file 相关联的“credit.dat”文件中。

## 15.9 向随机访问文件中随机地写入数据

图 15.12 所示的程序向文件“credit.dat”中写入数据，为了进行读和写，以“rw”方式打开该文件。程序中使用 RandomAccessFile 的 seek 方法，得到存储信息的记录在文件中的确切位置。seek 方法将文件位置指针定位到文件中相对于文件起始位置的指定位置，随后 Record 类的 write 方法输出这些数据。

```

1 // Fig. 15.12: WriteRandFile.java
2 // This program uses TextFields to get information from the
3 // user at the keyboard and writes the information to a
4 // random access file.
5 import java.io.*;
6 import java.awt.*;
7
8 public class WriteRandFile extends Frame {
9
10     // Application window components
11     TextField acct,          // where user enters account number
12         fName,              // where user enters first name

```

```

13         lName,          // where user enters last name
14         bal;            // where user enters balance
15     Button enter,        // send record to file
16         done;           // quit program
17     Label acctLabel,     // account label
18         fNameLabel,      // first name label
19         lNameLabel,      // last name label
20         balLabel;        // balance label
21
22     // Application other pieces
23     RandomAccessFile output; // file for output
24     Record data;
25
26     // Constructor -- initialize the Frame
27     public WriteRandFile()
28     {
29         super( "Write to random access file" );
30
31         data = new Record();
32
33         // Open the file
34         try {
35             output = new RandomAccessFile( "credit.dat", "rw" );
36         }
37         catch ( IOException e ) {
38             System.err.println( e.toString() );
39             System.exit( 1 );
40         }
41
42         setup();
43     }
44
45     public void addRecord()
46     {
47         int acctNum = 0;
48         Double d;
49
50         acctNum = ( new Integer( acct.getText() ) ).intValue();
51
52         // output the values to the file
53         try {
54             if ( acctNum > 0 && acctNum <= 100 ) {
55                 data.account = acctNum;
56                 data.firstName = fName.getText();
57                 data.lastName = lName.getText();
58                 d = new Double ( bal.getText() );
59                 data.balance = d.doubleValue();
60                 output.seek( (long) ( acctNum-1 ) * data.size() );
61                 data.write( output );
62
63                 // clear the TextFields
64                 acct.setText( "" );
65                 fName.setText( "" );
66                 lName.setText( "" );
67                 bal.setText( "" );

```

```
68         }
69         else {
70             acct.setText( "Enter valid account (1-100)" );
71             acct.selectAll();
72         }
73     }
74     catch ( IOException e ) {
75         System.err.println( "Error during write to file\n" +
76                             e.toString() );
77         System.exit( 1 );
78     }
79 }
80
81 // Setup the window for the application with TextFields
82 // and Buttons
83 public void setup()
84 {
85     resize( 300, 150 );
86     setLayout( new GridLayout( 5, 2 ) );
87
88     // create the components of the Frame
89     acct = new TextField( 20 );
90     acctLabel = new Label( "Account Number" );
91     fName = new TextField( 20 );
92     fNameLabel = new Label( "First Name" );
93     lName = new TextField( 20 );
94     lNameLabel = new Label( "Last Name" );
95     bal = new TextField( 20 );
96     balLabel = new Label( "Balance" );
97     enter = new Button( "Enter" );
98     done = new Button( "Done" );
99
100    // add the components to the Frame
101    add( acctLabel );    // add label
102    add( acct );         // add TextField
103    add( fNameLabel );   // add label
104    add( fName );        // add TextField
105    add( lNameLabel );   // add label
106    add( lName );        // add TextField
107    add( balLabel );     // add label
108    add( bal );          // add TextField
109    add( enter );        // add button
110    add( done );         // add button
111
112    show();              // show the Frame
113 }
114
115 // Cleanup--add the current information (if valid),
116 // flush remaining output to file and close file.
117 public void cleanup()
118 {
119     if ( !acct.getText().equals("") )
120         addRecord();
121
122     try {
123         output.close();
```



```

124     }
125     catch ( IOException e ) {
126         System.err.println( "File not closed properly\n" +
127                             e.toString() );
128         System.exit( 1 );
129     }
130 }
131
132 // Process the event when the user closes the window
133 // or presses the "Done" button
134 public boolean handleEvent( Event event )
135 {
136     // User closed the window or clicked Done button
137     if ( event.id == Event.WINDOW_DESTROY ||
138         event.target == done ) {
139         cleanup(); // write data, close file, etc.
140
141         hide();
142         dispose(); // release system resources
143         System.exit( 0 );
144         return true;
145     }
146
147     // User clicked Enter button to send record to file
148     if ( event.target instanceof Button ) {
149         if ( event.arg.equals( "Enter" ) ) {
150             addRecord();
151             return true;
152         }
153     }
154
155     return super.handleEvent( event );
156 }
157
158 // Instantiate a CreateSeqFile object and start the program
159 public static void main( String args[] )
160 {
161     WriteRandFile accounts = new WriteRandFile();
162 }
163 }

```

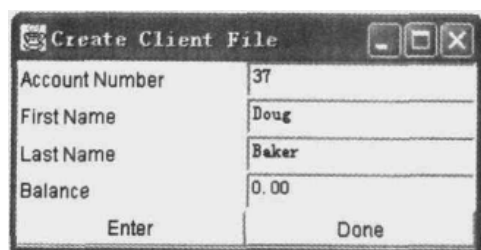


图 15.12 向随机访问文件中随机地写入数据

用户输入账号、姓氏、名字和余额，接着按下Enter按钮后，将引发WriteRandFile类的addRecord方法。该方法从文本字段中检索数据，并利用Record类的名为data的对象来存放数据，然后调用Record类的write方法输出数据。

下列语句：

```
output.seek( (long)(acctNum-1) * data.size() );
```

把用于 output 对象的文件位置指针定位到由“(long)(acctNum-1) \* data.size()”计算出的字节位置处。由于账号值在 1 和 100 之间，因此在计算记录的字节位置时，要将账号减 1。这样，对于记录 1，文件位置指针将设置为文件的 0 字节。这个计算结果的类型为 long，因为 seek 方法需要一个 long 值作为参数。

如果用户按下 Done 按钮，程序就会调用 WriteRandFile 类的 cleanup 方法向文件中加入最后一个记录（如果有一个记录正在等待输出），然后关闭文件并终止程序。该程序假定用户没有输入重复的账号，并且用户在每个文本字段都输入了合适的值。

## 15.10 从随机访问文件中顺序地读取数据

在前面几节，我们创建了一个随机访问文件并向该文件写入数据。在这一节，我们将开发一个程序（如图 15.13 所示），为了读文件，程序中使用“r”打开方式来打开一个 RandomAccessFile，顺序地读取完整个文件并只显示那些含有数据的记录。这个程序会产生一个额外的好处，我们将在本节最后指出。

```

1 // Fig. 15.13: ReadRandFile.java
2 // This program reads a random access file sequentially and
3 // displays the contents one record at a time in text fields.
4 import java.io.*;
5 import java.awt.*;
6
7 public class ReadRandFile extends Frame {
8
9     // Application window components
10    TextField acct,    // displays account number
11            fName,    // displays first name
12            lName,    // displays last name
13            bal;      // displays balance
14    Button next,      // display next record
15            done;     // quit program
16    Label acctLabel,  // account label
17            fNameLabel, // first name label
18            lNameLabel, // last name label
19            balLabel;  // balance label
20
21    // Application other pieces
22    RandomAccessFile input; // file from which data is read
23    Record data;
24    boolean moreRecords = true;
25
26    // Constructor -- initialize the Frame
27    public ReadRandFile()
28    {
29        super( "Read Client File" );
30
31        // Open the file
32        try {

```

```
33         input = new RandomAccessFile( "credit.dat", "r" );
34     }
35     catch ( IOException e ) {
36         System.err.println( e.toString() );
37         System.exit( 1 );
38     }
39
40     data = new Record();
41
42     setup();
43 }
44
45 // Read a record and display it on the screen
46 public void readRecord()
47 {
48     // read a record and display
49     try {
50         do {
51             data.read( input );
52             } while ( input.getFilePointer() < input.length() &&
53                     data.account == 0 );
54         }
55     catch( IOException e ) {
56         moreRecords = false;
57     }
58
59     if ( data.account != 0 ) {
60         acct.setText( String.valueOf( data.account ) );
61         fName.setText( data.firstName );
62         lName.setText( data.lastName );
63         bal.setText( String.valueOf( data.balance ) );
64     }
65 }
66
67 // Setup the window for the application with TextFields
68 // and Buttons
69 public void setup()
70 {
71     resize( 300, 150 );
72     setLayout( new GridLayout( 5, 2 ) );
73
74     // create the components of the Frame
75     acct = new TextField( 20 );
76     acctLabel = new Label( "Account Number" );
77     fName = new TextField( 20 );
78     fNameLabel = new Label( "First Name" );
79     lName = new TextField( 20 );
80     lNameLabel = new Label( "Last Name" );
81     bal = new TextField( 20 );
82     balLabel = new Label( "Balance" );
83     next = new Button( "Next" );
84     done = new Button( "Done" );
85
86     // add the components to the Frame
87     add( acctLabel );    // add label
88     add( acct );        // add TextField
```

```
89         add( fNameLabel );    // add label
90         add( fName );          // add TextField
91         add( lNameLabel );     // add label
92         add( lName );          // add TextField
93         add( balLabel );       // add label
94         add( bal );            // add TextField
95         add( next );           // add button
96         add( done );           // add button
97
98         show();                // show the Frame
99     }
100
101     // Cleanup--close the file.
102     public void cleanup()
103     {
104         try {
105             input.close();
106         }
107         catch ( IOException e ) {
108             System.err.println( e.toString() );
109             System.exit( 1 );
110         }
111     }
112
113     // Process the Next button
114     public boolean action( Event event, Object o )
115     {
116         // Determine if the next record should be displayed
117         if ( event.target instanceof Button )
118             if ( event.arg.equals( "Next" ) )
119                 readRecord();
120
121         return true;
122     }
123
124     // Process the event when the user closes the window
125     // or presses the done button
126     public boolean handleEvent( Event event )
127     {
128         // User closed the window or clicked Done
129         if ( moreRecords == false ||
130             event.id == Event.WINDOW_DESTROY ||
131             event.target == done ) {
132             cleanup(); // close file.
133
134             hide();
135             dispose(); // release system resources
136             System.exit( 0 );
137             return true;
138         }
139
140         return super.handleEvent( event );
141     }
142
143     // Instantiate a ReadSeqFile object and start the program
144     public static void main( String args[] )
145     {
```

```
146         ReadRandFile accounts = new ReadRandFile();  
147     }  
148 }
```

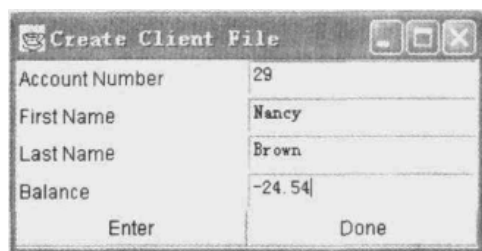


图 15.13 顺序地读取一个随机访问文件

#### 编程技巧 15.1

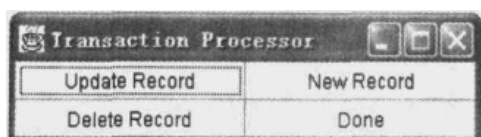
如果文件的内容不需要修改，就利用“r”文件打开方式打开文件以便读取，这样能够防止对文件内容的无意修改。这是最小使用权限原则的一个例子。

当用户按下Next按钮来读取文件中的下一个记录时，将会引发ReadRandFile类的readRecord方法。该方法将循环执行，并调用Record类的read方法来将数据输入到Record类的data对象。readRecord方法循环地读取“credit.dat”文件中的记录，直到读取到一个含有信息的记录为止。readRecord方法检查每个记录，并通过判断账号是否为0（所有记录的初值）来查看该记录是否包含数据。如果记录包含一个合法的账号（即非零值），则将终止循环，并且在文本字段中显示数据。如果用户按下Done按钮，就会引发cleanup方法来关闭文件和终止程序。如果在读取的过程中遇到文件结束标志，则将moreRecords标志置为false。当用户再次按下Next按钮时，程序将会终止。

而这个额外的好处是什么呢？在程序的执行过程中，如果读者检查窗口输出，就会注意到记录是按照排序过的顺序（按照账号）列出来的，这是我们采用直接访问技术将记录存入文件所获得的效果。这与我们已经讨论过的冒泡排序（参见第5章）相比，采用直接访问技术进行排序是非常快的。让文件尽可能大，以至于可以存放所创建的每一个记录，就可以达到这种排序速度。但是，这意味着文件在大多数时间内都是稀疏地存放数据，大量的空间就被浪费了。这样，我们得到另一个以空间换时间的例子：利用大量的空间，可以开发一个更快的排序算法。

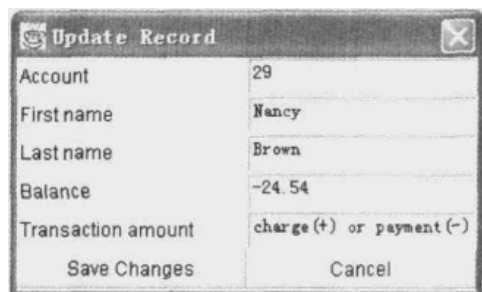
## 15.11 案例：交易处理程序

现在，我们给出一个交易处理系统的一部分，这个程序使用随机访问文件来达到即时访问处理的目的。该程序维护了一个银行的账户信息，可以更新现有的账户，添加新账户，删除账户，并为了打印目的将当前所有账户按照格式化的表格形式存放在一个文本文件中。我们假定已经执行了图15.11所示的程序，并创建了一个名为“credit.dat”的文件；而且也已经执行了图15.12所示的程序，并为记录加入了初值。该程序创建了4个按钮，可以选择不同任务，如下所示：



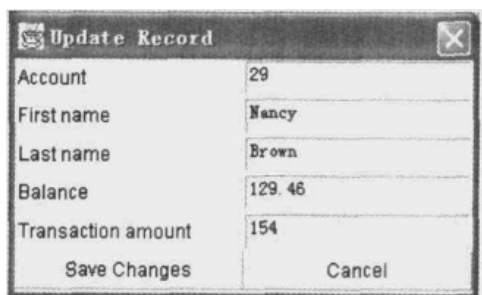
“Update Record”按钮显示“Update Record”对话框并更新一个账户。该方法只会更新一个现有的记录，因此用户必须首先输入一个账号并在键盘上按下回车键。UpdateRecord类的action方法

检查账号是否合法，然后使用 Record 类的 read 方法将记录读入 data 对象中。下一步，程序把 data.account 同 0 进行比较，以判断该记录是否包含信息。如果 data.account 是 0，将会在“Account”文本字段中显示一条信息，表明该记录不存在。如果证实该记录确实合法，信息将显示在如下所示的文本字段中：



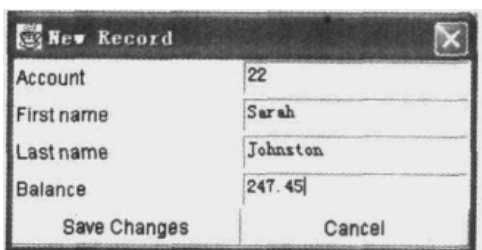
Account	29
First name	Nancy
Last name	Brown
Balance	-24.54
Transaction amount	charge (+) or payment (-)
Save Changes      Cancel	

我们注意到，“Transaction amount”文本字段中含有字符串“charge(+) or payment(-)”。用户通过选择这个文本字符串来输入交易金额（正值表示存款，负值表示取款），并在键盘上按下回车键来完成输入。通过这个操作方法可以得到交易金额，然后把它加到现有的余额上并在“Balance”文本字段中显示新的余额。用户可以按下“Save Change”按钮将更新过的记录写入到文件中，或者按下“Cancel”按钮取消交易，然后返回到主应用窗口。输入的交易结果如下所示：



Account	29
First name	Nancy
Last name	Brown
Balance	129.46
Transaction amount	154
Save Changes      Cancel	

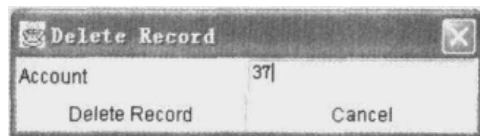
按下“New Record”按钮将显示“New Record”对话框，并将一个新账户加入到文件中。用户首先在“Account”文本字段中输入一个账号，并在键盘上按下回车键。NewRec 类的 action 方法检查该账号并判断它是否合法。如果合法，就将记录读入 data 对象。如果账号与现有的账号重复，就会在“Account”文本字段中显示一条信息，说明该账户已经存在。否则，提示用户在“First Name”、“Last Name”和“Balance”文本字段中输入数据。通过高亮显示每个字段的提示并在提示后输入相应的数据，就可以完成该操作。当数据输入完成后，可以按下“Save Changes”按钮将数据输出到文件中。一个典型的“New Record”操作的输出如下所示：



Account	22
First name	Sarah
Last name	Johnston
Balance	247.45
Save Changes      Cancel	

“Delete Record”按钮用来显示“Delete Record”对话框，并从文件中删除一条记录。用户在 Account 文本字段中输入账号，为了立即得到账户是否存在的信息，当用户按下回车键时将引发 DeleteRec 类的 action 方法。该方法将记录读入 data 对象并进行检查，以得到对象是否合法的信息。只有已经存在的记录才能删除，如果某账户为空，就会显示一条错误信息。在用户按下“Delete

Record”按钮后,就会将磁盘上的记录初始化,这样就删除了数据。一个典型的“Delete Record”操作的输出如下所示:



“Done”按钮用来终止程序运行。程序代码如图15.14所示,通过创建一个RandomAccessFile对象来打开文件“credit.dat”,打开方式为“rw”。

```

1  // Fig. 15.14: TransactionProcessor.java
2  // Transaction processing program using RandomAccessFiles.
3
4  // This program reads a random access file sequentially,
5  // updates data already written to the file, creates new
6  // data to be placed in the file, and deletes data
7  // already in the file.
8  import java.awt.*;
9  import java.io.*;
10
11 public class TransactionProcessor extends Frame {
12
13     // Application window components
14     Button updateButton, // update record
15         newButton,      // add new record
16         deleteButton,   // delete record
17         done;           // quit program
18
19     UpdateRec update;    // dialog box for record update
20     NewRec newRec;       // dialog box for new record
21     DeleteRec deleteRec; // dialog box for delete record
22
23     // Application other pieces
24     RandomAccessFile file; // file from which data is read
25     Record data;
26
27     // Constructor -- initialize the Frame
28     public TransactionProcessor()
29     {
30         super( "Transaction Processor" );
31
32         // Open the file
33         try {
34             file = new RandomAccessFile( "credit.dat", "rw" );
35         }
36         catch ( IOException e ) {
37             System.err.println( e.toString() );
38             System.exit( 1 );
39         }
40
41         data = new Record();
42
43         setup();
44     }

```

```
45
46     // Setup the window for the application with TextFields
47     // and Buttons
48     public void setup()
49     {
50         resize( 300, 80 );
51         setLayout( new GridLayout( 2, 2 ) );
52
53         updateButton = new Button( "Update Record" );
54         newButton = new Button( "New Record" );
55         deleteButton = new Button( "Delete Record" );
56         done = new Button( "Done" );
57
58         add( updateButton );
59         add( newButton );
60         add( deleteButton );
61         add( done );
62
63         show();           // show the Frame
64
65         // Create dialog boxes
66         update = new UpdateRec( file );
67         newRec = new NewRec( file );
68         deleteRec = new DeleteRec( file );
69     }
70
71     // Process actions
72     public boolean action( Event event, Object o )
73     {
74         if ( event.target instanceof Button ) {
75             String current = (String) event.arg;
76
77             if ( current.equals( "Update Record" ) )
78                 update.show();
79             else if ( current.equals( "New Record" ) )
80                 newRec.show();
81             else if ( current.equals( "Delete Record" ) )
82                 deleteRec.show();
83         }
84
85         return true;
86     }
87
88     // Process the event when the user closes the window
89     // or presses the "Done" button
90     public boolean handleEvent( Event event )
91     {
92         // User closed the window or clicked Done
93         if ( event.id == Event.WINDOW_DESTROY ||
94             event.target == done ) {
95             cleanup();
96
97             hide();
98             dispose(); // release system resources
99             System.exit( 0 );
100             return true;
```



```
101         }
102
103         // May need to process other events in action
104         return super.handleEvent( event );
105     }
106
107     public void cleanup()
108     {
109         try {
110             file.close();
111         }
112         catch ( IOException e ) {
113             System.err.println( e.toString() );
114             System.exit( 1 );
115         }
116     }
117
118     public static void main( String args[ ] )
119     {
120         TransactionProcessor teller = new TransactionProcessor();
121     }
122 }
123
124 class UpdateRec extends Dialog {
125     RandomAccessFile file;
126     TextField acct,
127         fName,
128         lName,
129         bal,
130         transaction;
131     Label acctLabel,
132         fNameLabel,
133         lNameLabel,
134         balLabel,
135         transLabel;
136     Button cancel,
137         save;
138
139     Record data;
140     int account;
141
142     public UpdateRec( RandomAccessFile f )
143     {
144         super( new Frame(), "Update Record", true );
145         resize( 300, 180 );
146         setLayout( new GridLayout( 6, 2 ) );
147         file = f;
148
149         acct = new TextField( 10 );
150         fName = new TextField( 10 );
151         lName = new TextField( 10 );
152         bal = new TextField( 10 );
153         transaction = new TextField( 10 );
154         acctLabel = new Label( "Account" );
155         fNameLabel = new Label( "First name" );
156         lNameLabel = new Label( "Last name" );
```

```
157         balLabel = new Label( "Balance" );
158         transLabel = new Label( "Transaction amount" );
159         save = new Button( "Save Changes" );
160         cancel = new Button( "Cancel" );
161
162         add( acctLabel );
163         add( acct );
164         add( fNameLabel );
165         add( fName );
166         add( lNameLabel );
167         add( lName );
168         add( balLabel );
169         add( bal );
170         add( transLabel );
171         add( transaction );
172         add( save );
173         add( cancel );
174
175         data = new Record();
176     }
177
178     public boolean action( Event event, Object o )
179     {
180         if ( event.target == acct ) {
181             account = Integer.parseInt( acct.getText() );
182
183             if ( account < 1 || account > 100 ) {
184                 acct.setText( "Invalid account" );
185                 return true;
186             }
187
188             try {
189                 file.seek( ( account - 1 ) * data.size() );
190                 data.read( file );
191             }
192             catch ( IOException e ) {
193                 acct.setText( "Error reading file" );
194             }
195
196             if ( data.account != 0 ) {
197                 acct.setText( String.valueOf( data.account ) );
198                 fName.setText( data.firstName );
199                 lName.setText( data.lastName );
200                 bal.setText( String.valueOf( data.balance ) );
201                 transaction.setText( "charge(+) or payment(-)" );
202             }
203             else
204                 acct.setText( String.valueOf( account ) +
205                             " does not exist" );
206         }
207         else if ( event.target == save ) {
208             try {
209                 file.seek( ( account - 1 ) * data.size() );
210                 data.write( file );
211             }
212             catch ( IOException e ) {
```

```
213         acct.setText( "Error writing file" );
214         return true;
215     }
216
217     hide();
218     clear();
219 }
220 else if ( event.target == cancel ) {
221     hide();
222     clear();
223 }
224 else if ( event.target == transaction ) {
225     Double trans =
226         Double.valueOf( transaction.getText() );
227     data.balance += trans.doubleValue();
228     bal.setText( String.valueOf( data.balance ) );
229 }
230
231     return true;
232 }
233
234 private void clear()
235 {
236     acct.setText( "" );
237     fName.setText( "" );
238     lName.setText( "" );
239     bal.setText( "" );
240     transaction.setText( "" );
241 }
242 }
243
244 class NewRec extends Dialog {
245     RandomAccessFile file;
246     TextField acct,
247         fName,
248         lName,
249         bal;
250     Label acctLabel,
251         fNameLabel,
252         lNameLabel,
253         balLabel;
254     Button cancel,
255         save;
256
257     Record data;
258     int account;
259
260     public NewRec( RandomAccessFile f )
261     {
262         super( new Frame(), "New Record", true );
263         resize( 300, 150 );
264         setLayout( new GridLayout( 5, 2 ) );
265         file = f;
266
267         acctLabel = new Label( "Account" );
268         acct = new TextField( 10 );
```

```
269         fName = new TextField( 10 );
270         lName = new TextField( 10 );
271         bal = new TextField( 10 );
272         fNameLabel = new Label( "First name" );
273         lNameLabel = new Label( "Last name" );
274         balLabel = new Label( "Balance" );
275         save = new Button( "Save Changes" );
276         cancel = new Button( "Cancel" );
277
278         add( acctLabel );
279         add( acct );
280         add( fNameLabel );
281         add( fName );
282         add( lNameLabel );
283         add( lName );
284         add( balLabel );
285         add( bal );
286         add( save );
287         add( cancel );
288
289         data = new Record();
290     }
291
292     public boolean action( Event event, Object o )
293     {
294         if ( event.target == acct ) {
295             account = Integer.parseInt( acct.getText() );
296
297             if ( account < 1 || account > 100 ) {
298                 acct.setText( "Invalid account" );
299                 return true;
300             }
301
302             try {
303                 file.seek( ( account - 1 ) * data.size() );
304                 data.read( file );
305             }
306             catch ( IOException e ) {
307                 acct.setText( "Error reading file" );
308             }
309
310             if ( data.account == 0 ) {
311                 fName.setText( "Enter first name" );
312                 lName.setText( "Enter last name" );
313                 bal.setText( "Enter balance" );
314             }
315             else {
316                 acct.setText( String.valueOf( data.account ) +
317                             " already exists" );
318                 fName.setText( "" );
319                 lName.setText( "" );
320                 bal.setText( "" );
321             }
322         }
323         else if ( event.target == save ) {
324             try {
```

```
325         data.account = account;
326         data.lastName = lName.getText();
327         data.firstName = fName.getText();
328         data.balance = ( Double.valueOf(
329             bal.getText() ) ).doubleValue();
330         file.seek( ( account - 1 ) * data.size() );
331         data.write( file );
332     }
333     catch ( IOException e ) {
334         acct.setText( "Error writing file" );
335         return true;
336     }
337
338     hide();
339     clear();
340 }
341 else if ( event.target == cancel ) {
342     hide();
343     clear();
344 }
345
346     return true;
347 }
348
349 private void clear()
350 {
351     acct.setText( "" );
352     fName.setText( "" );
353     lName.setText( "" );
354     bal.setText( "" );
355 }
356 }
357
358 class DeleteRec extends Dialog {
359     RandomAccessFile file;
360     TextField acct;
361     Label acctLabel;
362     Button cancel,
363         delete;
364
365     Record data;
366     int account;
367
368     public DeleteRec( RandomAccessFile f )
369     {
370         super( new Frame(), "Delete Record", true );
371         resize( 300, 80 );
372         setLayout( new GridLayout( 2, 2 ) );
373         file = f;
374
375         acctLabel = new Label( "Account" );
376         acct = new TextField( 10 );
377         delete = new Button( "Delete Record" );
378         cancel = new Button( "Cancel" );
379
380         add( acctLabel);
```

```
381         add( acct );
382         add( delete );
383         add( cancel );
384
385         data = new Record();
386     }
387
388     public boolean action( Event event, Object o )
389     {
390         if ( event.target == acct ) {
391             account = Integer.parseInt( acct.getText() );
392
393             if ( account < 1 || account > 100 ) {
394                 acct.setText( "Invalid account" );
395                 return true;
396             }
397
398             try {
399                 file.seek( ( account - 1 ) * data.size() );
400                 data.read( file );
401             }
402             catch ( IOException e ) {
403                 acct.setText( "Error reading file" );
404             }
405
406             if ( data.account == 0 )
407                 acct.setText( String.valueOf( account ) +
408                             " does not exist" );
409         }
410         else if ( event.target == delete ) {
411             try {
412                 file.seek( ( account - 1 ) * data.size() );
413                 data.account = 0;
414                 data.firstName = "";
415                 data.lastName = "";
416                 data.balance = 0.0;
417                 data.write( file );
418             }
419             catch ( IOException e ) {
420                 acct.setText( "Error writing file" );
421                 return true;
422             }
423
424             hide();
425             acct.setText( "" );
426         }
427         else if ( event.target == cancel ) {
428             hide();
429             acct.setText( "" );
430         }
431
432         return true;
433     }
434 }
```

图 15.14 交易处理程序

## 15.12 File 类

正如我们在本章开头所讲述的, java.io 软件包中包含大量处理输入和输出的类。我们已经集中讨论了以下的类: 处理顺序文件的类 ( `FileInputStream` 和 `FileOutputStream` ), 处理数据流的类 ( `DataInputStream` 和 `DataOutputStream` ), 以及处理随机访问文件的类 ( `RandomAccessFile` )。在这一节, 我们要介绍 `File` 类, 它对于检索文件或目录的磁盘信息特别有用。 `File` 类的对象并不能真正打开一个文件或提供任何文件操作功能。

使用 `File` 对象的一个例子是检查一个文件是否存在。在“常见编程错误 15.1”中, 我们曾经提示过, 如果为了输出而用 `FileOutputStream` 对象打开一个已经存在的文件, 就会在没有警告的情况下丢失该文件的内容。一个 `File` 对象可以用来判断一个文件已经存在。如果文件存在, 就可以使用 `RandomAccessFile` 对象代替 `FileOutputStream` 对象来打开该文件, 或者至少可以警告用户他们会丢掉文件原来的内容。

### 编程技巧 15.2

在使用 `FileOutputStream` 对象打开一个文件前, 应该使用 `File` 对象来判断该文件是否存在。

使用下面 3 个构造函数之一就可以初始化一个 `File` 对象。构造函数:

```
public File (String name)
```

将 `String` 参数 `name` 存放在一个对象中, `name` 可以包含文件或目录名的路径信息, 文件或目录的路径可以引导用户在磁盘上找到该文件或目录。路径包括指向文件或目录的全部或部分目录。一个绝对路径包括从根目录开始的所有目录, 它能找到一个指定的文件或目录。一个指定的磁盘驱动器中的每个文件或目录在它们的路径中都有相同的根目录。一个相对路径包括指向某个指定的文件或目录的目录集的子集, 相对路径以当前所操作的目录开始。

构造函数:

```
public File (String pathToName, String name)
```

使用参数 `pathToName` (绝对或相对路径) 来定位名为 `name` 的文件或目录。

构造函数:

```
public File (File directory, String name)
```

使用以前创建的 `File` 对象 `directory` (绝对或相对路径) 来定位名为 `name` 的文件或目录。

图 15.15 中给出了一些使用 `File` 类的常用 `public` 方法, 可以参看 Java API 来了解其他 `File` 方法。

方法	描述
<code>boolean canRead()</code>	如果文件可读则返回 <code>true</code> , 否则返回 <code>false</code>
<code>boolean canWrite()</code>	如果文件可写则返回 <code>true</code> , 否则返回 <code>false</code>
<code>boolean exists()</code>	如果作为参数传递给 <code>File</code> 构造函数的名称是指定路径下的文件或目录, 那么返回 <code>true</code> , 否则返回 <code>false</code>
<code>boolean isFile()</code>	如果作为参数传递给 <code>File</code> 构造函数的名称是一个文件, 那么返回 <code>true</code> , 否则返回 <code>false</code>
<code>boolean isDirectory()</code>	如果作为参数传递给 <code>File</code> 构造函数的名字是一个目录, 那么返回 <code>true</code> , 否则返回 <code>false</code>
<code>boolean isAbsolute()</code>	如果 <code>File</code> 构造函数的参数指明了一个文件或目录的绝对路径, 那么返回 <code>true</code> , 否则返回 <code>false</code>
<code>String getAbsolutePath()</code>	返回一个 <code>String</code> , 它表示文件或目录的绝对路径
<code>String getName()</code>	返回一个 <code>String</code> , 它表示文件或目录的名称

(续表)

方法	描述
<code>string getPath()</code>	返回一个 String, 它表示文件或目录的路径
<code>string getParent()</code>	返回一个 String, 它表示文件或目录的父目录——即在其中能找到该文件或目录的目录
<code>long length()</code>	返回用字节数表示的文件长度, 如果 File 对象表示一个目录, 则返回 0
<code>long lastModified()</code>	返回最后一次修改文件或记录的系统时间。返回的值只有在与利用该方法返回的其他值进行比较时才有用
<code>string[] list()</code>	返回一个表示目录内容的 String 数组

图 15.15 一些常用的 File 方法

图 15.16 中所示的应用程序示范了 File 类。FileTest 类创建了一个图形用户界面, 它包括 TextField enter 和 TextArea output。用户将文件或目录的名字输入到“enter”文本字段并按下回车键, FileTest 类的 action 方法创建一个新的 File 对象并给其分配名称。下一步, 测试 if 结构中的条件 (第 41 行) `name.exists()`。如果用户输入的文件名不存在, 那么 action 方法会前进到程序的第 80 行并输出用户输入的文件名, 后面紧跟着字符串 “does not exist”。否则, 程序将执行 if 结构体, 接着输出文件或目录的名称, 然后输出利用 `isFile`、`isDirectory` 和 `isAbsolute` 方法测试的 File 对象的结果。接着显示 `lastModified`、`length`、`getPath`、`getAbsolutePath` 和 `getParent` 方法的返回值。最后, 如果 File 对象代表一个文件, 则程序将读入文件的内容并将其显示在文本区域。由于我们使用的是 `RandomAccessFile` 对象, 因此能为读操作而打开该文件, 并用 `readLine` 方法 (第 64 行) 一次读入文件的一行。我们注意到, `RandomAccessFile` 对象是由 File 对象 `name` 初始化的 (第 59 行)。如果 File 对象代表一个目录, 程序就利用 File 类的方法 `list` 读入目录的内容, 并将其显示在文本区域中。

```

1 // Fig. 15.16: FileTest.java
2 // Demonstrating the File class.
3 import java.awt.*;
4 import java.io.*;
5
6 public class FileTest extends Frame {
7     File name;
8     TextField enter;
9     TextArea output;
10
11     public FileTest()
12     {
13         super( "Testing class File" );
14         setLayout( new BorderLayout() );
15         enter = new TextField(
16             "Enter file or directory name here", 40 );
17         output = new TextArea( 20, 30 );
18         add( "North", enter );
19         add( "Center", output );
20         resize( 400, 400 );
21         show();
22     }
23
24     public boolean handleEvent( Event e )
25     {
26         if ( e.id == Event.WINDOW_DESTROY ) {
27             hide();
28             dispose();

```



```
29         System.exit( 0 );
30     }
31
32     return super.handleEvent( e );
33 }
34
35 public boolean action( Event e, Object o )
36 {
37     output.setText( "" );
38
39     name = new File( o.toString() );
40
41     if ( name.exists() ) {
42         output.appendText(
43             name.getName() + " exists\n" +
44             ( name.isFile() ? "is a file\n" :
45                 "is not a file\n" ) +
46             ( name.isDirectory() ? "is a directory\n" :
47                 "is not a directory\n" ) +
48             ( name.isAbsolute() ? "is absolute path\n" :
49                 "is not absolute path\n" ) +
50             "Last modified: " + name.lastModified() +
51             "\nLength: " + name.length() +
52             "\nPath: " + name.getPath() +
53             "\nAbsolute path: " + name.getAbsolutePath() +
54             "\nParent: " + name.getParent() );
55
56         if ( name.isFile() ) {
57             try {
58                 RandomAccessFile r =
59                     new RandomAccessFile( name, "r" );
60
61                 output.appendText( "\n\n" );
62
63                 do {
64                     output.appendText( r.readLine() + "\n" );
65                 } while( r.getFilePointer() < r.length() );
66             }
67             catch( IOException e2 ) {
68             }
69         }
70         else if ( name.isDirectory() ) {
71             String dir[ ] = name.list();
72
73             output.appendText( "\n\nDirectory contents:\n");
74
75             for ( int i = 0; i < dir.length; i++ )
76                 output.appendText( dir[ i ] + "\n" );
77         }
78     }
79     else {
80         output.appendText( o.toString() +
81             " does not exist\n" );
82     }
83
84     return true;
85 }
```

```
86
87     public static void main( String args[ ] )
88     {
89         FileTest f = new FileTest();
90     }
91 }
```

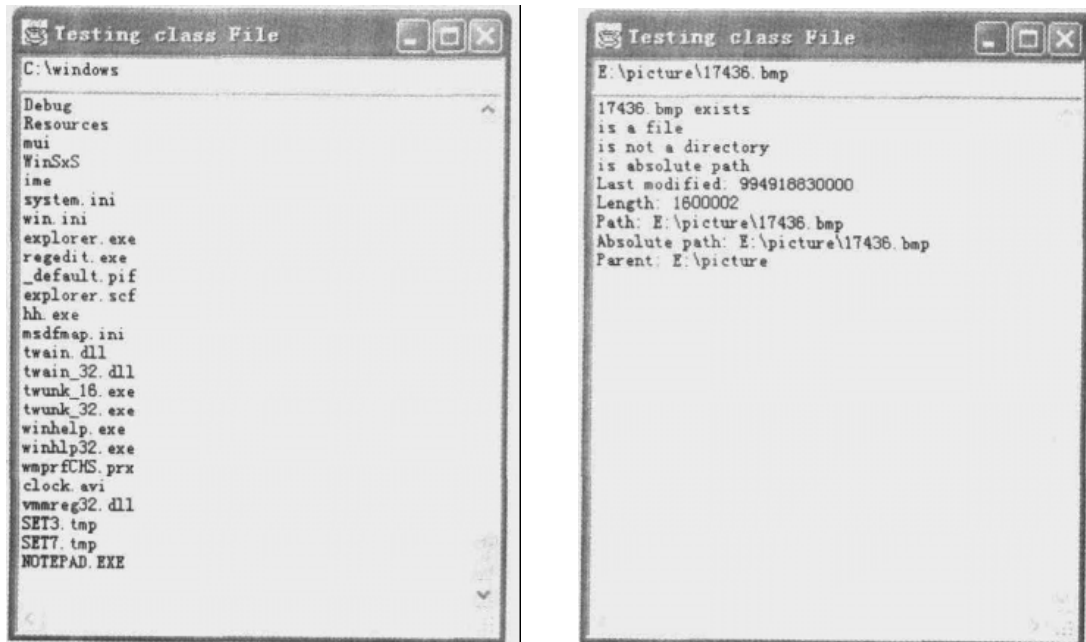


图 15.16 File 类

此程序的第一个输出示例了一个 File 对象，该对象与 C 盘下的 windows 目录相关联。第二个输出也示例了一个 File 对象，该对象与 E 盘下的 17436.bmp 文件相关联。在这两种情况中，都指定了我们自己的计算机中的绝对路径。注意，“\”分隔符分开路径中的文件和目录。在一台 UNIX 工作站上，分隔符会是“/”字符。实际上，Java 认为路径名中的这两种符号是一样的。因此，即使同时使用这两种分隔符描述路径 C:\java\README，Java 仍可以正确操作文件。

### 15.13 对象的输入/输出

在这一章讨论了 Java 面向对象的输入/输出风格，但是，我们的例子集中在传统数据类型而不是用户自定义类对象的 I/O 上。

如果是向一个磁盘文件输出对象的实例变量，那么在某种意义上将会丢失该对象的类型信息，在磁盘上将只有数据而没有类型信息。如果准备读取该数据的程序知道数据所对应的对象类型，那么就将数据简单地读入这种类型的对象。

当把不同类型的对象存储在同一个文件中时，就会出现一个有趣的问题。如果将它们同时读入一个程序，怎样才能区分它们（或它们的实例变量）呢？

一种方法是在收集表示一个对象的实例变量数据之前，让每个输出方法都输出一个类型代码。输入方法总是会从读取类型代码区开始，并使用 switch 语句来引发相应对象的输入方法。尽管这种解决方案缺少多态编程的优势，但它提供了一种利用文件保存对象并在需要时检索它们的可行机制。

## 小结

- 计算机操作的所有数据项最终都是0和1的组合。
- 计算机中最小的数据项是0和1, 这样的数据项称为一个位。
- 数字、字母和特殊符号统称为字符, 在某台计算机上编写程序和表示数据项的所有字符的集合称为这台计算机的字符集。字符集的每个字符都用1和0的序列来表示 (Java中的字符是2个字节的标准字符)。
- 一个域是表达一定含义的一组字符 (或字节)。
- 一条记录是一组相关的域。
- 一条记录中至少要有有一个域作为记录关键字, 它标识一条记录属于某个特定的人或实体, 并且该记录区别于文件中的所有其他记录。
- Java并不将结构强加于文件之上。这样, 类似“记录”这样的表示法就不存在于Java中, 程序员必须适当地组织文件的结构, 以符合特定应用程序的需求。
- 用来创建和管理数据库的程序的集合称为数据库管理系统 (DBMS)。
- Java把每一个文件都看成有序的字节流。
- 每个文件都使用一些与机器相关的文件结束符来终止。
- 流提供了文件和程序之间的通信通道。
- 我们必须将java.io软件包放入程序中才能进行Java文件的输入/输出, 这个软件包中包括诸如FileInputStream、FileOutputStream、DataInputStream和DataOutputStream等流类的定义。
- 可以使用流类FileInputStream、FileOutputStream和RandomAccessFile的实例对象来打开文件。
- InputStream (Object的子类) 和 OutputStream (Object的子类) 是抽象类, 它们分别定义了执行输入和输出操作的方法; 抽象类的派生类重载了这些方法。
- 文件的输入/输出是由FileInputStream (InputStream的子类) 和FileOutputStream (OutputStream的子类) 完成的。
- 管道是线程间的同步通信通道, 可以在两个线程间建立一个管道。一个线程通过对PipeOutputStream (OutputStream的子类) 进行写操作来向另一个线程传送数据; 目标线程通过PipeInputStream (InputStream的子类) 从管道中读取信息。
- PrintStream (FilterOutputStream的子类) 用来向屏幕 (或者由本地操作系统所定义的“标准输出”) 输出。事实上, 我们就是采用PrintStream来进行文本输出的; System.out就是一个PrintStream (System.err也是如此)。
- FilterInputStream过滤了InputStream而FilterOutputStream过滤了OutputStream; 简单来说, 过滤操作意味着过滤流提供了额外的功能, 诸如缓冲、监视行号或者将数据字节集中到原始数据类型的单元中等。
- 以原始字节的方式读取数据虽然速度很快, 但并不方便。通常情况下, 程序想以字节集合的形式, 例如整数、浮点数、双精度浮点数等形式来读取数据。为了达到这个目的, 我们需要使用DataInputStream (FilterInputStream的子类)。
- DataInput的接口由DataInputStream类和RandomAccessFile类 (下面将要讨论) 实现, 它们都需要从一个流中读取原始数据类型。
- DataInputStream能使程序从一个InputStream中读取二进制数据。
- DataInput的接口包括以下方法: read (对于字节数组)、readBoolean、readByte、readChar、

readDouble、readFloat、readFully（对于字节数组）readInt、readLine、readLong、readShort、readUnsignedByte、readUnsignedShort、readUTF（对于标准字符串）和 skipBytes。

- DataOutput 的接口由 DataOutputStream（FilterOutputStream 的子类）和 RandomAccessStream 类来实现，它们都需要向一个 OutputStream 中写入原始数据类型。
- DataOutputStream 能使程序向一个 OutputStream 中写入二进制数据。DataOutput 的接口包括以下方法：flush、size、write（对于单字节）、write（对于字节数组）、writeBoolean、writeByte、writeBytes、writeChar、writeChars（对于标准字符串）、writeDouble、writeFloat、writeInt、writeLong、writeShort 和 writeUTF。
- 设置缓冲是一种 I/O 操作增强技术。
- 通过一个 BufferedOutputStream（FilterOutputStream 的子类），输出指令就不会导致数据向输出设备进行实际的物理转换，而输出操作将直接面向称为缓冲区的一块内存区域。该区域足够大，能够容纳许多输出操作的数据。每当缓冲区全部占满以后，一个大的物理输出操作会完成对输出设备的实际输出，这种直接面向内存中输出缓冲区的操作通常称为逻辑输出操作。
- 通过一个 BufferedInputStream 类（FilterInputStream 的子类），一个大的物理输入操作将许多逻辑数据块从文件读入内存缓冲区。当程序需要新的数据块时，便从缓冲区中取得数据块（有时这也称为逻辑输入操作）。当缓冲区变空时，输入设备的下一个实际的物理输入操作会读取下一组“逻辑”数据块。这样一来，实际的物理输入操作的次数与程序发出的读请求的次数相比就会小多了。
- 利用 BufferOutputStream，可以在任何时候强制一个未占满的缓冲区通过一个外部的 flush 方法向设备输出。
- PushBackInputStream 类是 FilterInputStream 的一个子类，这种读一个 PushBackInputStream 的应用程序从流中读取若干字节，并且形成由几个字节组成的集合。有时，为了判断一个集合是否结束，应用程序不得不读取前一个集合“结束之后”的第一个字符。一旦程序断定当前的集合已经结束，就会将多读的字符送回到流中。
- 许多程序使用 PushBackInputStream 类，例如编译器就利用其来解析输入的数据，也就是把输入的数据分解成有意义的单元（例如 Java 编译器必须识别的关键字、标识符和运算符）。
- StreamTokenizer 类（Object 的一个子类）有助于将一个来自文本输入文件的文本流分解成称为标记的有意义的单元。
- 对于直接访问的应用程序，例如像航班预订系统和销售点系统这样的交易处理应用程序，RandomAccessFile 类（Object 的一个子类）是有用的。
- 对于顺序访问文件，每一个连续的输入/输出请求都读取或写入文件中下一个连续的数据集合。
- 对于随机访问文件，每一个连续的输入/输出请求都会指向文件的任意部分，可能与文件中上一次请求所引用的位置相距很远。
- 直接访问的应用程序提供了快速访问大文件中指定数据的功能；当人们正在等待回答并且这些答案很快就要用到的时候，当人们变得不耐烦并且要“换个地方做生意”的时候，这样的应用程序就会经常使用了。
- 使用 ByteArrayInputStream 类（InputStream 抽象类的子类）可以从内存中的字节数组输入。
- 使用 ByteArrayOutputStream 类（OutputStream 抽象类的子类）可以向内存中的字节数组输出。
- 字节数组的 I/O 应用程序完成对数据的确认。程序一次从输入流向字节数组输入完整的一行，

然后确认例程仔细检查字节数组的内容,并且在需要的时候更正数据。在知道了输入数据具有正确的格式之后,程序才能继续从字节数组输入。

- `StringBufferInputStream` 类 (`InputStream` 抽象类的子类) 从 `StringBuffer` 对象中得到输入。
- `SequenceInputStream` 类 (`InputStream` 抽象类的子类) 能够连接几个 `InputStream`, 以便程序把这些流看成一个连续的 `InputStream`。在每个输入流结束时关闭该流, 然后打开序列中的下一个流。
- `LineNumberInputStream` 类 (`FilterInputStream` 类的子类) 总能知道正在读取的文件的行号。
- 多数 Java 用户一般不使用 `FileDescriptor` 类的对象。
- `File` 类能够使程序获得一个文件或目录的信息。
- 通过创建一个 `FileOutputStream` 类的对象来打开一个文件以便输出。将一个参数——文件名传递给构造函数, 已经存在的文件将被截尾——文件中所有的数据都被丢弃了。如果这个指定的文件不存在, 那么就创建一个新文件。
- 对于无文件、一个文件和几个文件这些情况, 程序都能进行处理。每个文件都有一个同文件流对象相联系的惟一的名字。所有的文件操作方法都必须利用适当的对象指向一个文件。
- 文件位置指针指示下一个输入开始的位置或者下一个输出在文件中放置的位置。
- 实现一个随机访问文件的简便的办法是只使用固定长度的记录。利用这种技术, 程序能够迅速计算出一条记录相对于文件开始位置的确切位置。
- 我们可以不破坏文件中其他数据而将数据插入到随机访问文件中, 也可以不重写整个文件而更新或删除数据。
- `RandomAccessFile` 类具有 `DataInputStream` 和 `DataOutputStream` 类所具有的全部输入和输出功能, 它也能使用 `seek` 方法来查找文件中某个指定的字节位置。

## 术语

absolute path	绝对路径	character field	字符域
alphabetic field	字母域	character set	字符集
binary digit	二进制数字	close a file	关闭一个文件
bit	位	close method	close 方法
buffer	缓冲区	data hierarchy	数据层次
BufferedInputStream class	BufferedInputStream 类	database	数据库
BufferedOutputStream class	BufferedOutputStream 类	database management system(DBMS)	数据库管理系统 (DBMS)
buffering	设置缓冲	DataInput interface	DataInput 接口
byte	字节	DataInputStream class	DataInputStream 类
ByteArrayInputStream class	ByteArrayInputStream 类	DataOutput interface	DataOutput 接口
ByteArrayOutputStream class	ByteArrayOutputStream 类	DataOutputStream class	DataOutputStream 类
canRead method of File class	File 类的 canRead 方法	data validation	数据合法性
canWrite method of File class	File 类的 canWrite 方法	decimal digit	十进制数字
chaining stream objects	链接流对象	direct-access applications	直接访问的应用程序
		directory	目录
		end-of-file	文件结束

- end-of-file marker 文件结束符
- exists method of File class File 类的 exists 方法
- field 域
- file 文件
- File class File 类
- FileDescriptor class FileDescriptor 类
- FileInputStream class FileInputStream 类
- file name 文件名
- file-position pointer 文件位置指针
- FileInputStream class FileInputStream 类
- FileOutputStream class FileOutputStream 类
- FilterInputStream class FilterInputStream 类
- FilterOutputStream class FilterOutputStream 类
- flush
- getAbsolutePath method of File class File 类的  
getAbsolutePath 方法
- getName method of File class File 类的 getName  
方法
- getParent method of File class File 类的 getParent  
方法
- getPath method of File class File 类的 getPath  
方法
- input stream 输入流
- InputStream class InputStream 类
- instant-access application 即时访问的应用程序
- IOException
- isAbsolute method of File class File 类的 isAbsolute  
方法
- isDirectory method of File class File 类的 isDirectory  
方法
- isFile method of File class File 类的 isFile 方法
- lastModified method of File class File 类的  
lastModified 方法
- list method of File class File 类的 list 方法
- length method of File class File 类的 length 方法
- LineNumberInputStream class  
LineNumberInputStream 类
- logical input operation 逻辑输入操作
- logical output operation 逻辑输出操作
- memory buffer 内存缓冲区
- numeric field 数字域
- open a file 打开一个文件
- output stream 输出流
- OutputStream class OutputStream 类
- partially filled buffer 未填满的缓冲区
- persistent data 永久数据
- physical input operation 物理输入操作
- physical output operation 物理输出操作
- pipe 管道
- PipedInputStream class PipedInputStream 类
- PipedOutputStream class PipedOutputStream 类
- PrintStream class PrintStream 类
- PushBackInputStream class PushBackInputStream 类
- r file open mode “r” 文件打开方式
- random-access file 随机访问文件
- RandomAccessFile class RandomAccessFile 类
- read method read 方法
- readBoolean
- readByte
- readChar
- readDouble
- readFloat
- readFully
- readInt
- readLine
- readLong
- readShort
- readUnsignedByte
- readUnsignedShort
- readUTF
- record 记录
- record key 记录关键字
- relative path 相对路径
- root directory 根目录
- rw file open mode “rw” 文件打开方式
- SequenceInputStream class SequenceInputStream 类
- sequential-access file 顺序访问文件
- size
- skipBytes stream skipBytes 流
- standard output 标准输出
- StreamTokenizer class StreamTokenizer 类
- StringBufferInputStream class

StringBufferInputStream 类	writeBoolean
System.err(standard error stream) System.err (标准错误流)	writeByte
	writeBytes
System.in(standard input stream) System.in (标准输入流)	writeChar
	writeChars
System.out(standard output stream) System.out (标准输出流)	writeDouble
	writeFloat
token 标记	writeInt
transaction-processing systems 交易处理系统	writeLong
truncate an existing file 截尾一个已经存在的文件	writeShort
Unicode character set 标准字符集	writeUTF
write method write 方法	

## 自测练习

### 15.1 填空：

- 计算机操作的所有数据项最终都是 \_\_\_\_\_ 和 \_\_\_\_\_ 的组合。
- 计算机能处理的最小的数据项称为 \_\_\_\_\_。
- 一个 \_\_\_\_\_ 是一组相关的记录。
- 数字、字母和特殊符号统称为 \_\_\_\_\_。
- 一组相关的文件称为 \_\_\_\_\_。
- 文件流类 FileOutputStream、FileInputStream 和 RandomAccessFile 的 \_\_\_\_\_ 方法关闭一个文件。
- DataInputStream 类的 \_\_\_\_\_ 方法从指定的流中读取一个整数。
- DataInputStream 类的 \_\_\_\_\_ 方法从指定的流中读取一个字符串。
- RandomAccessFile 类的 \_\_\_\_\_ 方法在文件中将文件位置指针设置到某一指定的位置，以便输入和输出。

### 15.2 判断下面句子是否正确。如果不正确，请解释原因。

- 程序员必须从外部创建 System.in、System.out 和 System.err。
- 在一个顺序文件中，如果文件位置指针要指向一个文件开始位置以外的地方，那么就必须关闭该文件，然后再重新打开它并从文件开始位置进行读取。
- 在随机访问文件中，不用搜索全部记录就可以找到一个指定的记录。
- 随机访问文件中所有记录的长度都必须一致。
- seek 方法必须搜索相对于文件开始位置的位置。

### 15.3 假定下面每一条语句都在同一个程序中应用：

- 编写一条语句，打开文件 “oldmast.dat” 以便输入；使用与 FileInputStream 对象链接的 DataInputStream 的对象 inOldMaster。
- 编写一组语句，打开文件 “trans.dat” 以便输入；使用与 FileInputStream 对象链接的 DataInputStream 的对象 inTransaction。
- 编写一组语句，打开文件 “newmast.dat” 以便输出；使用与 FileOutputStream 对象链接的 DataOutputStream 的对象 outNewMaster。

- d) 编写一组语句, 从文件 “oldmast.dat” 中读取一条记录, 该记录包含整数 accountNum、字符串 name 和浮点指针 currentBalance; 使用 DataInputStream 的对象 inOldMaster。
- e) 编写一组语句, 从文件 “trans.dat” 中读取一条记录, 该记录包含整数 accountNum 和浮点指针 dollarAmount; 使用 DataInputStream 的对象 inTransaction。
- f) 编写一组语句, 向文件 “newmast.dat” 输出一条记录, 该记录包含整数 accountNum、字符串 name 和浮点指针 currentBalance; 使用 DataOutputStream 的对象 outNewMaster。
- 15.4 找出下面每个句子的错误, 并将其改正。

- a) DataOutputStream 类的对象 outPayable 所引用的文件 “payables.dat” 尚未打开。

```
outPayable.writeInt ( account );
outPayable.writeUTF ( company );
outPayable.writeDouble ( amount );
```

- b) 下面的语句应该从文件 “payables.dat” 中读一条记录。FileInputStream 类的对象 inPayable 引用这个文件, 并且 DataInputStream 类的对象 inReceivable 引用文件 “receivables.dat”。

```
account = inReceivable.readInt ( );
company = inReceivable.readUTF ( );
amount = inReceivable.readDouble ( );
```

## 自测练习答案

- 15.1 a) 1, 0。b) 位。c) 文件。d) 字符。e) 数据库。f) close。g) readInt。h) readUTF。i) seek。
- 15.2 a) 不正确。这三条流可以自动创建。
- b) 正确。
- c) 正确。
- d) 不正确。随机访问文件中的记录在通常情况下其长度是一致的。
- e) 正确。

- 15.3 a)

```
DataInputStream inOldMaster;
inOldMaster = new DataInputStream (
    new FileInputStream ( "oldmast.dat" ) );
```

- b)

```
DataInputStream inTransaction;
inTransaction = new DataInputStream (
    new FileInputStream ( "trans.dat" ));
```

- c)

```
DataOutputStream outNewMaster;
outNewMaster = new DataOutputStream (
    new FileOutputStream ( "newmast.dat" ));
```

- d)

```
accountNum= inOldMaster.readInt ( );
name = inOldMaster.readUTF ( );
currentBalance = inOldMaster.readDouble ( );
```



- e)
- ```
accountNum = inTransaction.readInt();
dollarAmount = inTransaction.readDouble();
```
- f)
- ```
outNewMaster.writeInt(accountNum);
outNewMaster.writeUTF(name);
outNewMaster.writeDouble(currentBalance);
```
- 15.4 a) 错误：向流输出数据之前没有打开文件“payables.dat”。
- 改正：创建一个新的与FileOutputStream类对象链接的DataOutputStream类对象，它打开“payables.dat”以便输出。
- b) 错误：使用不正确的DataInputStream类对象来从文件“payables.dat”中读取一条记录。
- 改正：使用对象inPayable来引用“payables.dat”。

## 练习

- 15.5 填空：
- 计算机在二级存储设备上把大量的数据存储在\_\_\_\_\_。
  - \_\_\_\_\_由若干个域组成。
  - 一个只包含数字、字母和空格的域称为一个\_\_\_\_\_域。
  - 为了帮助从文件中恢复指定的记录，每条记录的一个域将作为\_\_\_\_\_。
  - 计算机系统中存储的大多数信息都存储在\_\_\_\_\_文件中。
  - 一组表达一定含义的相关字符称为一个\_\_\_\_\_。
  - 标准流对象是\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- 15.6 判断下面句子是否正确。如果不正确，请说明原因。
- 计算机完成的各种功能在本质上就是对0和1的处理。
  - 人们喜欢处理位而不喜欢处理字符和域，这是因为位更简洁。
  - 人们使用字符来描述程序和数据项，然后计算机按照0和1的序列来处理和操作这些字符。
  - 一个人的6位邮政编码是数字域的一个例子。
  - 在计算机应用中，一个人的街道地址通常作为一个字母域。
  - 随着我们从域到字符，再到位等一步步的发展，计算机操作的数据形成了一个数据层次，其中的数据项变得越来越大，结构也越来越复杂。
  - 一个记录关键字之所以能够识别出一条记录，是因为关键字属于某个特定的域。
  - 许多组织将他们的所有信息都存储在一个文件中，以方便计算机的操作。
  - 在Java程序中，处理一个文件的每一条语句都从外部通过名字来引用该文件。
  - 当一个程序创建一个文件时，计算机为了以后的引用将会自动保留该文件。
- 15.7 练习15.3让读者编写一系列的语句。实际上，这些语句形成了一种重要的文件处理程序的核心，即文件匹配程序。在商业数据处理中，每个应用系统拥有几个文件是很普遍的。例如，在一个应收账款系统中，通常会有一个主文件，它包括每个顾客的详细信息，诸如顾客的姓名、地址、电话、应收账款、贷款限额、贴现、合约安排等，可能还会有最近的购买和现金支付的历史情况。当完成一笔交易的时候（即销售完成且收到现金付

款), 这些信息就会进入一个文件。在每个业务周期的最后(有些公司是一个月, 一些公司是一个星期, 有些情况下是一天), 主文件(练习 15.3 中称为“oldmast.dat”)就处理交易文件(练习 15.3 中称为“trans.dat”), 这样就会更新每个账户的购买和支付记录。在更新过程中, 主文件将重写为一个新文件(“newmast.dat”), 然后, 在下一个业务周期的最后, 就利用这个新文件再开始一次更新。文件匹配程序必须解决在单个文件中不存在的一些问题。例如, 匹配并不总是存在。主文件中的一个顾客在目前的业务周期中可能还没有任何购买和现金支付活动, 因此交易处理文件中也就不会出现这个顾客的记录。类似地, 一个有购买和现金支付活动的顾客可能刚进入系统, 公司还来不及为这个顾客创建一个主记录。

利用练习 15.3 中编写的语句作为基础, 创建一个完整的应收账款文件匹配程序。使用每个文件中的账号作为记录关键字来完成匹配。假定每个文件都是顺序文件并且按照账号递增的顺序存储记录, 当一次匹配发生时(即相同账号的记录同时出现在主文件和交易文件中), 应把交易文件中的金额总数加到主文件中的目前余额上, 并且写入“newmast.dat”的记录(假定交易文件中用正数表示购买, 负数表示付款)。如果某个账户有主记录而没有相应的交易记录, 则只需把主记录写入“newmast.dat”。如果有交易记录而没有主记录, 就打印信息“Unmatched transaction record for account number...”(填入交易记录中的账号)。

- 15.8 在编写完练习 15.7 的程序之后, 再创建一个简单的程序, 提供一些测试数据以便检查练习 15.7 的程序。使用下面的账号数据:

主文件

账号	姓名	差额
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

交易文件

账号	差额
100	27.14
300	62.11
400	100.56
900	82.17

利用练习 15.8 创建的测试数据文件运行练习 15.7 的程序。打印新的主文件。检查账户是否已正确地进行了更新。

- 15.9 可能有一些交易记录含有相同的记录关键字(实际上很常见)。之所以会发生这种情况, 是因为某个顾客在一个业务周期中可以进行多次购买和现金支付。重新编写在练习 15.7 中完成的应收账款文件匹配程序, 以便能够处理含有相同记录关键字的一些记录。修改练习 15.8 中的测试数据, 以包含下面的交易记录:

账号	美元总数
300	83.89
700	80.78
700	1.53

15.10 编写一组语句来完成下面的每个要求。假定下列结构：

```
class Person{
    char lastName [15];
    char firstName [15];
    char age [2];
}
```

已经定义，并且已经正确打开了随机访问文件。

- a) 使用 100 个包括 lastName = "unassigned"、firstName = "" 和 age = "0" 的记录来初始化文件 "nameage.dat"。
  - b) 输入 10 个姓氏、名字和年龄，并将它们写入文件。
  - c) 更新一条含有信息的记录，如果没有信息，就通知用户 "No info"。
  - d) 通过重新初始化一条含有信息的记录的方法来删除该记录
- 15.11 假设读者是一个五金工具仓库的所有者，需要保留一份清单，其中记录了仓库中有多少种不同的工具，每种工具的量有多少，以及每种的价格是多少。编写一个程序，使用 100 个空记录来初始化随机访问文件 "hardware.dat"，允许输入每件工具的数据，并列出全部工具；还可以删除不再拥有的工具记录，并允许用户更新文件中的任何信息。工具标识号应该是记录号，利用下列信息作为文件的初始内容：

记录号	工具名称	数量	单价
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jigsaw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

- 15.12 修改第4章的电话号码产生程序，使其能把输出写入一个文件，这样就可以便于用户随时读取该文件。如果能使用计算机化的词典，那么修改本程序，使它能在词典中查询七字母的单词。这个程序将创建一些有趣的七字母的组合，有时会包含两个或者更多的单词，例如，电话号码 8432677 产生 "THEBOSS"。修改程序以使用计算机中的词典来检查每个可能的七字母单词，看看它是否是一个正确的一字母单词后面紧跟着一个正确的六字母单词，或者是一个二字母单词后面紧跟着一个合法的五字母单词，依次类推。

# 第16章 网 络

## 教学目标

- 理解 Java 的网络成员
- 实现 Java 的网络应用
- 理解如何实现互相通信的 Java 客户/服务器结构
- 理解如何实现基于网络的相互协作的应用
- 能够编写“可以在网络上漫游的程序”
- 创建一个多线程的服务器

## 16.1 简介

Internet 和 WWW 是令人激动的, Internet 将“信息世界”联系在一起, WWW 则由于应用了多媒体技术而使得 Internet 更加容易使用, 各个团体都将 Internet 和 WWW 视为他们信息系统战略的关键。Java 早期的实现提供了一些内嵌的网络功能, 使用它们可以方便地开发基于 Internet 和 WWW 的应用程序。Java 不仅可以通过多线程来实现并发操作, 还可以使程序搜寻全世界的信息, 并同网络中其他计算机上运行的程序一同协作, 而无论这些机器是否在同一个组织、同一个国家或是在全球的任意位置上。Java 甚至可以使在同一台机器上运行的 Java applet 和应用程序互相通信。

网络是一个内容丰富且复杂的课题。Java 提供了丰富的网络功能, 并且可以用于计算机网络的教学实验。本书介绍了 Java 基本的网络概念和功能。我们提供了 5 个活动代码的例子, 帮助用户建立自己的各种较大规模的网络应用。Java 的网络功能都包含在 java.net 软件包中, 读者应该深入了解该软件包中的各种类和方法。

Java 使用基于套接字的网络通信, 这种方式使得用户可以采用同文件 I/O 一样的处理方式来看网络, 程序可以像读写文件那样对一个套接字 (socket) 进行操作。我们将在后面演示如何创建及使用套接字。

Java 提供了流套接字和数据报套接字。一个进程使用流套接字来同其他进程建立一个连接。当连接建立后, 数据流就可以在一个进程与另一个进程之间进行交换, 因此将流套接字称为提供面向连接的服务。这种传输协议就是通常所说的传输控制协议 (TCP, Transmission Control Protocol)。

可以使用数据报协议来传输单个信息分组。这种协议并不是我们通常使用的, 且不同于 TCP 协议, 我们将这种协议称为用户数据报协议 (UDP, User Datagram Protocol)。UDP 是一种无连接的协议, 也就是在任何情况下该协议都会不保证将数据报送到目的地。实际上, 数据报可能丢失, 或者被复制, 甚至不按顺序送到。因此, 如果采用 UDP, 用户端必须采用其他的复杂程序来处理这些问题。流套接字和 TCP 协议将成为绝大多数 Java 程序员的首选。

### 性能提示 16.1

面向非连接的服务将带来更好的实时性能, 但是这与面向连接的服务相比, 它的可靠性也就更差。

### 可移植性提示 16.1

TCP 协议以及与之相关的协议集得到了各种不同的计算机系统（采用不同处理器和操作系统）的支持。

我们有关网络的讨论将同时关注客户-服务器双方。客户方要求一些动作得到执行，而服务器方则执行这些动作，然后将结果（如果有）返回给客户。客户首先试图同服务器建立一个连接。服务器可以接受或拒绝这一请求。如果接受连接，那么客户同服务器之间就可以像使用文件 I/O 一样通过套接字进行通信。我们的网络例程将使用在第 15 章讨论过的流操作来进行输入/输出。

## 16.2 利用 URL

Internet 支持许多协议。超文本传输协议 (HTTP, HyperText Transfer Protocol) 形成了 WWW 使用 URL (Uniform Resource Locator, 统一资源定位器) 在 Internet 上进行数据定位的基础。如果知道 WWW 上任何站点的 URL, 就可以通过 HTTP 协议来得到其中的数据。Java 可以使读者轻松地使用 URL, 只要将 URL 作为 AppletContext 类中 showDocument 方法的一个参数, 就可以打开相应的 URL。

图 16.1 所示的程序允许用户通过点击一个按钮来访问相应的 WWW 站点。这个 applet 包含两个按钮——“Java Home”和“Deitel Home”。当用户点击其中某一个按钮时，SiteSelector 的方法将判断按下了哪一个按钮，然后得到同那个按钮相关联的 URL，并且显示它所对应的 WWW 主页。

```

1 // Fig. 16.1: SiteSelector.java
2 // This program uses a button to load a document from a URL.
3 import java.awt.*;
4 import java.net.*;
5 import java.applet.Applet;
6
7 public class SiteSelector extends Applet {
8     Site siteList[]; // array of sites
9
10    public void init()
11    {
12        siteList = new Site[ 2 ];
13        siteList[ 0 ] = new Site ( " Java Home ",
14                                   "http://www.javasoft.com/index.html" );
15        siteList[ 1 ] = new Site ( "Deitel Home",
16                                   "http://www.deitel.com/index.html" );
17
18        for ( int i = 0; i < siteList.length; i++ )
19            add( new Button( siteList[ i ].getTitle() ) );
20    }
21
22    public boolean action( Event e, Object arg )
23    {
24        if ( e.target instanceof Button ) {
25            String title;
26            URL location;
27
28            for ( int i = 0; i < siteList.length; i++ ) {
29                title = siteList [ i ].getTitle();
30                location =
31                    siteList[ i ].getLocation();
32            }
33        }
34    }
35}

```

```

33         if ( title.equals( arg.toString() ) ) {
34             gotoSite( location );
35             return true; // event handled
36         }
37     }
38 }
39
40     return false; // event not handled yet
41 }
42
43     public void gotoSite( URL loc )
44     {
45         // this must be executed in a browser such as Netscape
46         getAppletContext().showDocument( loc );
47     }
48 }
49
50     class Site extends Button {
51         private String title;
52         private URL location;
53
54         public Site( String siteTitle, String siteLocation )
55         {
56             title = siteTitle;
57
58             try {
59                 location = new URL( siteLocation );
60             }
61             catch ( MalformedURLException e ) {
62                 System.err.println( "Invalid URL: " + siteLocation );
63             }
64         }
65
66         public String getTitle() { return title; }
67
68         public URL getLocation1() { return location; }
69 }

```

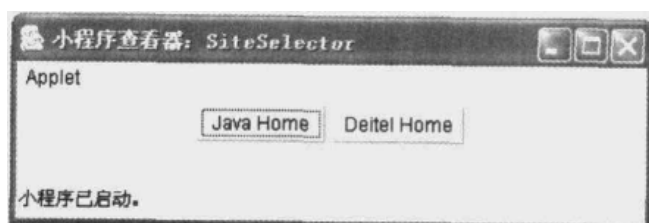


图 16.1 将文档从 URL 上载入浏览器

SiteSelector 类包含 Site 对象的一个数组。Site 类（在第 50 行定义）的每一个对象包含两个实例变量——一个 String 类型的 title 和一个 URL 类型的 location。当这个 applet 的 init 方法创建一个 Site 对象后，两个 String 将作为参数传递给构造函数。第一个 String 直接设置为 String title。第二个传递给类 URL 的构造函数，它需要一个 String 类型的参数。URL 的构造函数判断这个 String 是否代表一个合法的 URL 地址。如果合法，则将这个 URL 对象初始化为包含这个统一资源定位器；否则将抛出一个 MalformedURLException 异常。注意，URL 的构造函数必须在一个 try 程序块中调用，或者这

个 `MalformedURLException` 异常必须在 `Site` 的构造函数的 `throws` 子句中进行声明。

#### 常见编程错误 16.1

当传递给 `URL` 构造函数的一个 `String` 不是合法的 `URL` 格式时，将抛出一个 `MalformedURLException` 异常。

在这个 applet 的 `init` 方法创建完两个 `Site` 对象之后，第 18 行的 `for` 结构将创建两个 `Button` 对象。每一个按钮利用其所对应的 `Site` 的 `title` 来标识，`title` 是通过调用 `Site` 的方法 `getTitle` 来确定的。

当用户点击其中的某一个按钮时，将调用 `action` 方法。`for` 结构（第 28 行）将对象数组中每一个 `Site` 的标题同我们点击的按钮的标签进行比较，如果某个 `title` 与这个按钮的标签相同，此时将激活 applet 的 `gotoSite` 方法，传递新的 `URL` 地址并将其显示。

下列语句：

```
getAppletContext().showDocument( loc );
```

在 `gotoSite` 方法中使用了 `getAppletContext` 方法（从 `Applet` 类中继承而来）来得到一个 `AppletContext` 对象的引用，这个对象代表了此 applet 的环境——即执行这个 applet 的浏览器。然后这个引用将激活 `AppletContext` 类的 `showDocument` 方法，它将一个 `URL` 的对象作为参数接收进来，然后在浏览器中显示与这个 `URL` 相关的 `WWW` 资源。在本例中，两个资源都是 `WWW` 的站点。

注意，同这个 applet 相关联的外部窗口用来显示 appletviewer 中执行的这个 Java applet。这个 applet 必须在一个像 Netscape Navigator 这样的浏览器中执行，才能观察到在结果中显示的另一个 `WWW` 主页的信息。appletviewer 仅仅具有执行一个 Java applet 的功能，因此它所用到的 HTML 文件中的其他部分都将忽略。如果我们在程序中用到的 `WWW` 主页包含 Java applet，那么在点击某一个按钮时，在 appletviewer 中将只显示包含的 Java applet。

## 16.3 采用 URL 的流连接从服务器上读取文件

图 16.2 的程序采用了 `URL` 对象，在服务器的一个文件中打开 `InputStream`，然后将此文件的内容读出并在 applet 的文本字段中显示。这个 `ReadServerFile` 类包含一个称为 `fileURL` 的 `URL` 对象，它将由一个指向测试文件（在 Deitel & Associates 公司的站点上）的统一资源定位器进行初始化，这个测试文件（“test.txt”）包含一个示例文本。

```
1 // Fig. 16.2: ReadServerFile.java
2 // This program uses a URL connection to read a file
3 // on the server.
4 import java.awt.*;
5 import java.net.*;
6 import java.io.*;
7 import java.applet.Applet;
8
9 public class ReadServerFile extends Applet {
10     URL fileURL;
11     TextArea contents;
12     InputStream input;
13     DataInputStream dataInput;
14
15     public void init()
```

```
16     {
17         contents = new TextArea( "Please wait...", 10, 40 );
18         add( contents );
19
20         try {
21             fileURL = new URL(
22                 "http://www.deitel.com/test/test.txt" );
23         }
24         catch ( MalformedURLException e ) {
25             showStatus( "Exception: " + e.toString() );
26         }
27     }
28
29     public void start()
30     {
31         String text;
32
33         try {
34             input = fileURL.openStream();
35             dataInput = new DataInputStream( input );
36             contents.setText( "The file contents are:\n" );
37
38             while ( ( text = dataInput.readLine() ) != null )
39                 contents.appendText( text + "\n" );
40
41             dataInput.close();
42         }
43         catch ( IOException e ) {
44             showStatus( "Exception: " + e.toString() );
45         }
46     }
47 }
```

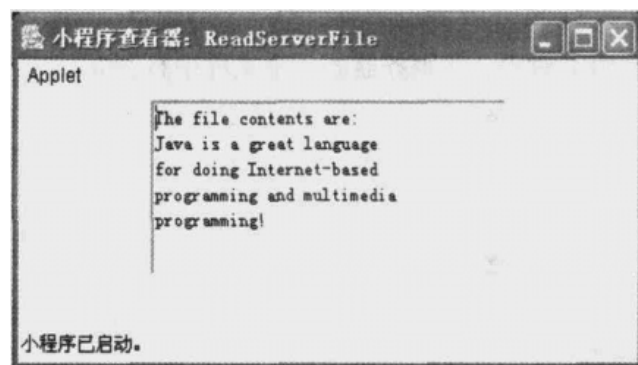


图 16.2 通过 URL，打开连接来读取文件

这个 applet 的 `init` 方法创建一个文本字段并显示它，然后创建同测试文件相关联的 `URL` 对象。`start` 方法使用下面的语句，打开一个同服务器上的文件相关联的流：

```
input = fileURL.openStream() ;
```

这条语句使用 `URL` 的 `openStream` 方法，取得一个同服务器上的文件相关联的 `InputStream` 对象，然后将它赋给引用 `input`。接下来，我们把一个 `DataInputStream` 同 `InputStream` 对象的输入进行关联，



这样就能使用 `DataInputStream` 的方法 `readLine` 来从文件中读取一行文本。

第 38 行的 `while` 结构从文件中一次读取一行文本，并将 `dataInput.readLine()` 返回的 `String` 赋给引用 `text`，然后在文本字段后面添加显示出 `String` 类型的 `text`。当整个文件都读取完时，使用 `dataInput.close()` 关闭输入流。

## 16.4 建立一个简单的服务器（采用流套接字）

为了建立一个简单的服务器，我们创建一个 `ServerSocket` 和一个 `Socket`。这个服务器登记一个有效的端口号（port number），客户将在这个端口向服务器请求连接。`ServerSocket` 建立这个服务器用来等待客户请求的端口，每一个连接由一个 `Socket` 管理。同时，需要创建一个 `OutputStream`，服务器利用它来发送数据；一个 `InputStream`，服务器使用它来接收数据。然后采用下面的语句来登记服务：

```
ServerSocket s = new ServerSocket ( port , queueLength ) ;
```

`queueLength` 规定了服务器能够处理的等待连接的客户数目。如果队列满了，将自动拒绝客户的请求。一旦建立起 `ServerSocket`，服务器将无限等待客户的连接，这是通过调用 `ServerSocket` 的方法来完成的：

```
Socket connection = s.accept( ) ;
```

当一个建立起连接以后，上述语句将返回一个 `Socket` 对象。

### 软件工程视点 16.1

通过使用套接字，Java 程序中的网络 I/O 看起来很像文件 I/O。对于程序员来说，套接字将许多网络编程的复杂特性隐藏起来。

### 软件工程视点 16.2

通过使用 Java 的多线程，我们能够创建可以处理多个客户同时进行多个请求的多线程服务器；这种多线程服务器体系结构在 UNIX、Windows NT 和 OS/2 的网络服务器上得到了广泛应用。

### 软件工程视点 16.3

多线程服务器的实现方法可以是为每一个 `accept` 方法调用返回的 `Socket` 创建一个新的线程，从而处理其上的网络 I/O；或者，多线程服务器的实现可以通过维护线程缓冲区，从而对一个新建的 `Socket` 的网络 I/O 进行管理。

### 性能提示 16.2

在内存充足的、具有高性能要求的系统中，一个多线程服务器可以创建一个线程缓冲池。这样，每当一个新 `Socket` 建立起来时，系统都能立即处理相应的网络 I/O。因此，当建立起一个连接之后，服务器就不需要重新创建一个线程来处理它，也就不必耗费创建线程的时间。

服务器激活 `Socket` 中的 `getOutputStream` 方法，从而得到和 `Socket` 相关联的 `OutputStream` 的一个引用。如果有必要，服务器还可以激活 `Socket` 中的 `getInputStream` 方法，以得到和 `Socket` 相关联的 `InputStream` 的一个引用。`OutputStream` 和 `InputStream` 的对象能够分别通过 `OutputStream` 的 `write` 方法及 `InputStream` 的 `read` 方法，从而进行单一字节或字节集合的发送或接收。通常，发送和接收 `int`、`double` 和 `String` 这样类型的数据要比简单的字节更加方便。在这种情况下，我们能够采用第 15 章中提到的技术，将其他的流类型（例如 `DataOutputStream` 和 `DataInoutStream`）与同 `Socket` 相关联的

OutputStream 和 InputStream 进行链接。例如:

```
DataInputStream input =  
    new DataInputStream( connection.getInputStream() );  
DataOutputStream output =  
    new DataOutputStream( connection.getOutputStream() );
```

建立这种关系的好处在于, 无论服务器写到 DataOutputStream 中的是什么内容, 都将通过 OutputStream 发送信息, 并且客户将在 InputStream 中得到这些信息; 同时无论客户写到 OutputStream (可能采用了一个相应的 DataOutputStream) 中的是什么内容, 都可以通过服务器的 InputStream 得到相应的信息。

另外一个常见的发送信息的方法是通过一个 PrintStream 链接到一个 OutputStream 上。可以按照以下的步骤实现:

```
PrintStream output =  
    new PrintStream( connection.getOutputStream() );
```

当信息的传输完毕后, 服务器通过 Socket 的方法 close 来关闭连接。

## 16.5 建立一个简单的客户 (采用流套接字)

为了建立一个客户, 我们使用一个 Socket 和服务器进行连接, 使用一个 InputStream 来接收服务器发送来的信息, 并且使用一个 OutputStream (如果必须) 将信息发回服务器。使用下面的这条语句, 可以建立同服务器的连接:

```
Socket connection = new Socket( "serverName" , port );
```

这条语句返回一个 Socket。如果一个连接请求失败, 则将抛出一个 IOException 异常。

### 常见编程错误 16.2

当客户请求的服务器地址无法得到正确的解析时, 将抛出一个 UnknownHostException 异常。

getInputStream 方法和 getOutputStream 方法分别用来得到与 Socket 相关的 InputStream 和 OutputStream。InputStream 的方法 read 用来从服务器输入单一的字节或字节集合。OutputStream 的方法 write 用来向服务器输出单一的字节或字节集合。正如我们在前面的章节提到的, 通常发送和接收 int、double 和 String 这样类型的数据要比简单的字节更加方便。如果服务器发送的信息采用实际的数据类型, 客户也将得到实际格式的信息。因此, 如果服务器使用一个 DataOutputStream 来发送信息, 客户应该使用一个 DataInputStream 来读取这些值。

当处理服务器发来的信息时, 客户必须判断服务器何时结束信息发送, 以便客户调用 close 方法来关闭 Socket 的连接。例如, InputStream 的 read 方法返回 -1 时, 表明读到流结束符 (也称为文件结束符: EOF, end-of-file)。如果 DataInputStream 用来从服务器读入信息, 那么当用户读到流结束符之后还想从流中读取信息时 (使用不同于 readLine 的一个方法), 则将会抛出一个 EOFException 异常。readLine 方法在读取到流结束符时将返回 null (如图 16.2 所示)。

当客户关闭 Socket 时, 可能会抛出一个 IOException 异常。getInputStream 方法和 getOutputStream 同样可能抛出 IOException 异常。

## 16.6 通过流套接字进行的客户/服务器交互

图16.3和图16.4所示的程序使用了流套接字来示例一个客户/服务器的应用。客户端的应用连接到服务器上,服务器端的应用发送数据到客户端,然后客户将接收到的数据显示出来。Server类的定义如图16.3所示,Client类的定义如图16.4所示。

Server类从Frame类扩展而来,所以需要提供一个构造函数,设置用于显示的窗口图形组件。Server的对象将输出显示到一个文本字段中,并且,handleEvent方法用来保证用户在应用结束时关闭相关的窗口。

Server类的runServer方法(第21行)用来设置服务器,以便接收一个连接请求,并在收到后处理该请求。这个方法声明了一个称为server的ServerSocket来等待连接请求,一个称为connection的Socket处理来自客户的连接,同时一个称为output的OutputStream通过connection来向客户发送数据。在try程序块中,将ServerSocket设置成用来监听在端口15000申请连接的客户请求。小于1024的端口号是保留给系统的,通常不能在用户程序中自定义为连接端口。构造函数的第二个参数是可以在队列中等待的、与服务器连接的最大数目(本例中为100)。如果队列已满但又收到新的请求,则将自动拒绝这个请求。

```
1 // Fig. 16.3: Server.java
2 // Set up a Server that will receive a connection
3 // from a client, send a string to the client,
4 // and close the connection.
5 import java.io.*;
6 import java.net.*;
7 import java.awt.*;
8
9 public class Server extends Frame {
10     TextArea display;
11
12     public Server()
13     {
14         super( "Server" );
15         display = new TextArea( 20, 5 );
16         add( "Center", display );
17         resize( 300, 150 );
18         show();
19     }
20
21     public void runServer()
22     {
23         ServerSocket server;
24         Socket connection;
25         OutputStream output;
26
27         try {
28             server = new ServerSocket( 5000, 100 );
29             connection = server.accept();
30             display.setText( "Connection received...\n" );
31             display.appendText( "Sending data...\n" );
32             output = connection.getOutputStream();
33             String s = new String( "Connection successful\n" );
34
```

```
35         for ( int i = 0; i < s.length(); i++ )
36             output.write( (int) s.charAt( i ) );
37
38         display.appendText(
39             "Transmission complete. Closing socket.\n" );
40         connection.close();
41     }
42     catch ( IOException e ) {
43         e.printStackTrace();
44     }
45 }
46
47 public boolean handleEvent( Event e )
48 {
49     if ( e.id == Event.WINDOW_DESTROY ) {
50         hide();
51         dispose();
52         System.exit( 0 );
53     }
54
55     return super.handleEvent( e );
56 }
57
58 public static void main( String args[] )
59 {
60     Server s = new Server();
61
62     s.runServer();
63 }
64 }
```

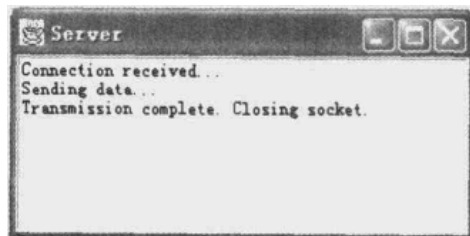


图 16.3 演示通过流套接字进行的客户/服务器连接的服务器部分

下列语句：

```
connection = server.accept() ;
```

使用 `ServerSocket` 的方法 `accept` 来等待一个连接。这个方法直到接收一个连接时才能终止。一旦接收了连接，处理这个连接的 `Socket` 对象将赋给 `connection`。下列语句：

```
output = connection.getOutputStream();
```

将与 `connection` 相关联的 `OutputStream` 的引用赋给 `output`。

第 35 行的 `for` 结构使用 `output`，将 `String` “`Connection successful\n`” 中的单个字符发送给客户。注意，每一个字符将强制转换为整数。这是因为 `OutputStream` 的方法 `write` 要求一个整数作为参数。

当传输结束后，使用下面的语句来闭 Socket：

```
connection.close() ;
```

注意，main 方法（第 58 行）创建一个 Server 类的对象，并在其中激活 runServer 方法。还要注意，Server 接收一个连接，处理这个连接并且最后终止执行。另一个非常相似的方案是 Server 接收一个连接，将这个要处理的连接设置为一个单独执行的线程，然后等待新的连接请求。在 Server 关注新的连接请求的同时，这个单独执行的线程将继续处理已有的连接。

```

1      // Fig. 16.4: Client.java
2      // Set up a Client that will read information sent
3      // from a Server and display the information.
4      import java.io.*;
5      import java.net.*;
6      import java.awt.*;
7
8      public class Client extends Frame {
9          TextArea display;
10
11         public Client()
12         {
13             super( "Client" );
14             display = new TextArea( 20, 10 );
15             add( "Center", display );
16             resize( 300, 150 );
17             show();
18         }
19
20         public void runClient()
21         {
22             Socket client;
23             InputStream input;
24
25             try {
26                 client = new Socket( InetAddress.getLocalHost(),
27                                     5000 );
28                 display.appendText( "Created Socket\n" );
29
30                 input = client.getInputStream();
31                 display.appendText( "Created input stream\n" );
32
33                 display.appendText(
34                     "The text from the server is:\n\t");
35                 char c;
36
37                 while ( ( c = (char) input.read() ) != '\n' )
38                     display.appendText( String.valueOf( c ) );
39
40                 display.appendText( "\n" );
41                 client.close();
42             }
43             catch ( IOException e ) {
44                 e.printStackTrace();
45             }

```

```
46     }
47
48     public boolean handleEvent( Event e )
49     {
50         if ( e.id == Event.WINDOW_DESTROY ) {
51             hide();
52             dispose();
53             System.exit( 0 );
54         }
55
56         return super.handleEvent( e );
57     }
58
59     public static void main( String args[] )
60     {
61         Client c = new Client();
62
63         c.runClient();
64     }
65 }
```

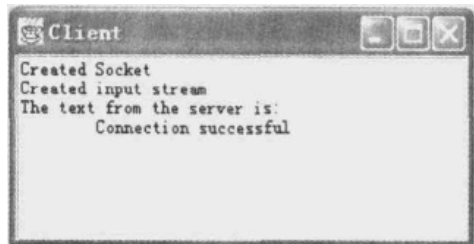


图 16.4 演示通过流套接字进行的客户/服务器连接的客户部分

同 Server 类相似, Client 类从 Frame 类扩展而来, 所以需要提供构造函数来设置用于显示的窗口。Client 类将输出显示到一个文本字段中, 同样, handleEvent 方法用来保证用户在应用结束时关闭相关的窗口。

Client 的方法 runClient 处理同服务器 Server 的连接并从那里接收数据的必要工作。在 try 块中, Socket client 将两个参数——InetAddress.getLocalHost() 和 5000 传递给构造函数。第一个参数返回一个 InetAddress 对象, 该对象包含程序正在其上执行的本地主机名。在本例中, 我们通过连接同一主机上的程序来演示客户/服务器关系。通常, 第一个参数应该是 Internet 上另一台计算机的地址。第二个参数是服务器的端口号, 这个数值应该同服务器正在等待接收服务的端口号完全相同(有时我们也称其为握手点, handshake point)。

一旦连接建立, 将在文本字段中显示一条消息, 并且将执行下列语句:

```
iinput = client.getInputStream();
```

以得到同 Socket 类的 client 相关联的 InputStream 的引用, 并将其赋给 input。引用 input 用来处理从 Server 发送到 Client 的数据。第 37 行的 while 结构使用了 InputStream 的 read 方法, 从流中一次读取一个字节。注意, 这个字节将强制转换为 char 类型, 这样它可以作为一个字符输出。在本例中, 我们知道 Server 传输的数据采用一个换行符(\n)结束, 因此我们可以明确地在 while 结构的条件判断中检查这个字符, 从而确定是否结束循环。当不再需要使用 Socket 来得到信息时, 我们使用下面的语句将其关闭:

```
client.close() ;
```

注意, 本例中的 Server 仅仅发送信息, 而 Client 仅仅接收信息。当然, 在一个客户/服务器体系中, 客户方与服务器方都可以发送和接收信息。

## 16.7 采用数据报方式进行无连接的客户/服务器交互

我们已经讨论了面向连接的、基于流的传输方式。现在我们考虑采用数据报方式来进行无连接的传输。

面向连接的传输更像我们使用的电话系统: 拨出一个号码并同想要通话一方的电话机建立一个连接; 在使用的过程中这个连接将一直保持, 哪怕用户并没有在说话。

采用数据报的无连接传输方式更像通过邮局进行邮件传递。如果一份邮件太大以至于一个信封放不下, 则可以将它分成几部分, 然后把它们分别放入单独的、已编号的信封中。实际上一次只寄出一封信。这些信件可能按照编号的顺序到达, 也可能不按顺序到达, 甚至有的信件将会丢失(尽管这样的情况很少见, 但毕竟存在这样的可能性)。接收方在读取信息之前, 将它们重新排好序。如果信息量不大并且能够放入一个信封之中, 当然不需要担心“顺序混乱”的问题。但是, 仍然可能由于某种差错而导致信件无法送达。

图 16.5 和图 16.6 所示程序在客户应用和服务器应用之间使用数据报来发送信息包。

```
1 // Fig. 16.5: Server.java
2 // Set up a Server that will receive packets from a
3 // client and send packets to a client.
4 import java.io.*;
5 import java.net.*;
6 import java.awt.*;
7
8 public class Server extends Frame {
9     TextArea display;
10
11     DatagramPacket sendPacket, receivePacket;
12     DatagramSocket sendSocket, receiveSocket;
13
14     public Server()
15     {
16         super( "Server" );
17         display = new TextArea( 20, 10 );
18         add( "Center", display );
19         resize( 400, 300 );
20         show();
21
22         try {
23             sendSocket = new DatagramSocket();
24             receiveSocket = new DatagramSocket( 5000 );
25         }
26         catch( SocketException se ) {
27             se.printStackTrace();
28             System.exit( 1 );
29         }
30     }
31 }
```

```
32     public void waitForPackets()
33     {
34         while ( true ) {
35             try {
36                 // set up packet
37                 byte array[] = new byte[ 100 ];
38                 receivePacket =
39                     new DatagramPacket( array, array.length );
40
41                 // wait for packet
42                 receiveSocket.receive( receivePacket );
43
44                 // process packet
45                 display.appendText( "\nPacket received:" +
46                     "\nFrom host: " + receivePacket.getAddress() +
47                     "\nHost port: " + receivePacket.getPort() +
48                     "\nLength: " + receivePacket.getLength() +
49                     "\nContaining:\n\t" );
50                 byte data[] = receivePacket.getData();
51                 String received = new String( data, 0 );
52                 display.appendText( received );
53
54                 // echo information from packet back to client
55                 display.appendText( "\n\nEcho data to client...");
56                 sendPacket = new DatagramPacket( data, data.length,
57                     receivePacket.getAddress(), 5001 );
58                 sendSocket.send( sendPacket );
59                 display.appendText( "Packet sent\n" );
60             }
61             catch( IOException exception ) {
62                 display.appendText( exception.toString() + "\n" );
63                 exception.printStackTrace();
64             }
65         }
66     }
67
68     public boolean handleEvent( Event e )
69     {
70         if ( e.id == Event.WINDOW_DESTROY ) {
71             hide();
72             dispose();
73             System.exit( 0 );
74         }
75
76         return super.handleEvent( e );
77     }
78
79     public static void main( String args[] )
80     {
81         Server s = new Server();
82
83         s.waitForPackets();
84     }
85 }
```



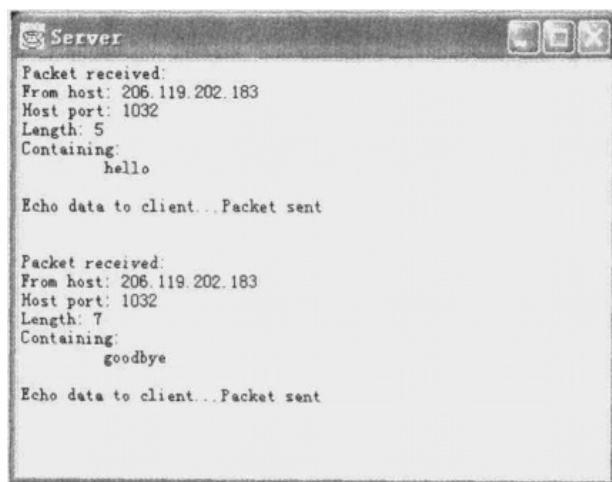


图 16.5 采用数据报方式进行无连接的客户/服务器交互的服务器端

```

1 // Fig. 16.6: Client.java
2 // Set up a Client that will send packets to a
3 // server and receive packets from a server.
4 import java.io.*;
5 import java.net.*;
6 import java.awt.*;
7
8 public class Client extends Frame {
9     TextField enter;
10    TextArea display;
11    Panel enterPanel;
12    Label enterLabel;
13
14    DatagramPacket sendPacket, receivePacket;
15    DatagramSocket sendSocket, receiveSocket;
16
17    public Client()
18    {
19        super( "Client" );
20        enterPanel = new Panel();
21        enterLabel = new Label( "Enter message:" );
22        enter = new TextField( 20 );
23        enterPanel.add( enterLabel );
24        enterPanel.add( enter );
25        add( "North", enterPanel );
26        display = new TextArea( 20, 10 );
27        add( "Center", display );
28        resize( 400, 300 );
29        show();
30
31        try {
32            sendSocket = new DatagramSocket();
33            receiveSocket = new DatagramSocket( 5001 );
34        }
35        catch( SocketException se ) {
36            se.printStackTrace();

```

```
37         System.exit( 1 );
38     }
39 }
40
41 public void waitForPackets()
42 {
43     while ( true ) {
44         try {
45             // set up packet
46             byte array[] = new byte[ 100 ];
47             receivePacket =
48                 new DatagramPacket( array, array.length );
49
50             // wait for packet
51             receiveSocket.receive( receivePacket );
52
53             // process packet
54             display.appendText( "\nPacket received:" +
55                 "\nFrom host: " + receivePacket.getAddress() +
56                 "\nHost port: " + receivePacket.getPort() +
57                 "\nLength: " + receivePacket.getLength() +
58                 "\nContaining:\n\t" );
59             byte data[] = receivePacket.getData();
60             String received = new String( data, 0 );
61             display.appendText( received );
62         }
63         catch( IOException exception ) {
64             display.appendText( exception.toString() + "\n" );
65             exception.printStackTrace();
66         }
67     }
68 }
69
70 public boolean handleEvent( Event e )
71 {
72     if ( e.id == Event.WINDOW_DESTROY ) {
73         hide();
74         dispose();
75         System.exit( 0 );
76     }
77
78     return super.handleEvent( e );
79 }
80
81 public boolean action( Event event, Object o )
82 {
83     try {
84         display.appendText( "\nSending packet containing: " +
85             o.toString() + "\n" );
86         String s = o.toString();
87         byte data[] = new byte[ 100 ];
88         s.getBytes( 0, s.length(), data, 0 );
89         sendPacket = new DatagramPacket( data, s.length(),
90             InetAddress.getLocalHost(), 5000 );
```

```

91         sendSocket.send( sendPacket );
92         display.appendText( "Packet sent\n" );
93     }
94     catch ( IOException exception ) {
95         display.appendText( exception.toString() + "\n" );
96         exception.printStackTrace();
97     }
98
99     return true;
100 }
101
102 public static void main( String args[] )
103 {
104     Client c = new Client();
105
106     c.waitForPackets();
107 }
108 }

```

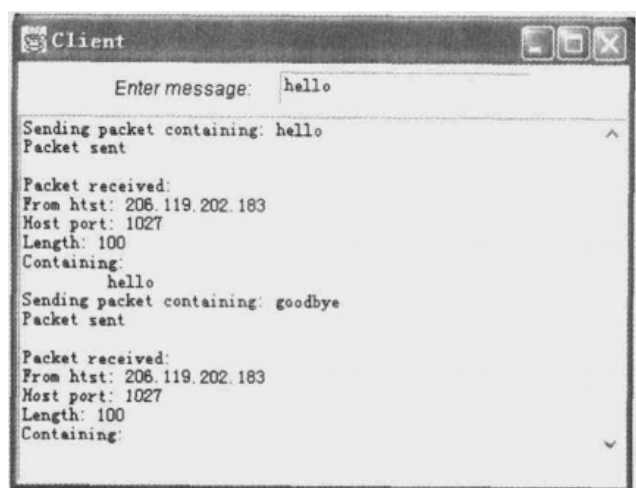


图 16.6 采用数据报方式进行无连接的客户 / 服务器交互的客户端

在图 16.6 的程序中, 用户在客户应用的文本字段中输入一条信息然后按下回车键, 这条信息将转化为一个 `byte` 类型的数组, 然后将其置于一个数据报中发送给服务器。服务器接收到数据报后, 将其中的信息显示出来, 然后将这个数据报返回给客户。当客户接收到该信息后, 同样将数据报中的信息显示出来。在这个程序中, `Client` 和 `Server` 两个类的实现是相似的。

`Server` 类 (如图 16.5 所示) 定义了两个 `DatagramPacket`, 它们用来创建发送和接收信息的数据报; 还定义了两个 `DatagramSocket`, 它们用来发送和接收这些数据报。`Server` 类的构造函数首先创建用来显示数据报内容的图形用户界面, 然后在 `try` 块中创建 `DatagramSocket`。下列语句:

```
sendSocket = new DatagramSocket();
```

使用了不带参数的 `DatagramSocket` 构造函数, 设置在网络上发送数据报的套接字。计算机将为 `sendSocket` 对象自动分配一个可用的端口号。下列语句:

```
receiveSocket = new DatagramSocket( 5000 );
```

使用带有一个整数端口号作为参数的 `DatagramSocket` 构造函数, 设置从网络上接收数据报的端口。

我们已经指定端口号为 5000。Client 使用指定的端口号 5000 向 Server 传输数据。如果设置 DatagramSocket 的某一个构造函数的执行失败，程序将抛出一个 SocketException 异常。

#### 常见编程错误 16.3

在创建一个 DatagramSocket 时，指定一个正在使用的端口号或者指定一个不合法的端口号都将导致一个 SocketException 异常。

Server 的 waitForPackets 方法将一直循环，等待到达 Server 一方的数据报。首先，该方法创建一个用来放置接收到的数据报内容的 DatagramPacket（第 37 行~第 39 行）。然后，执行下列语句：

```
receiveSocket.receive( receivePacket );
```

等待到达 Server 的数据报。receive 方法将一直等待，直到一个数据报到达并将此数据报置于其参数中（receivePacket）。

一旦数据报到达目的地，程序的第 45 行~第 52 行将输出这个数据报的内容。DatagramPacket 的 getAddress 方法返回一个包含这个数据报来源的主机名的 InetAddress 对象；getPort 方法返回这个数据报通过哪个端口进行传输的整数端口值；getLength 方法返回一个整数代表的发送数据按字节计算的长度；getData 方法返回包含发送数据的字节数组，这个字节数组在我们的程序中用来初始化一个 String，以便将此内容显示在文本字段中。

其次，实例化 sendPacket（用来把数据返回给客户），并将 4 个参数传递给 DatagramPacket 的构造函数。第一个参数指定将发送 byte 这个数组，第二个参数指定将发送的字节数，第三个参数指定这个数据报将发送到的客户地址，第四个参数指定客户在哪个端口等待接收数据报。下列语句：

```
sendSocket.send( sendPacket );
```

将数据报发送到网络上。无论发送方还是接收方，如果出现错误都将抛出一个 IOException 异常。

Client 类（如图 16.6 所示）同 Server 类是相似的，区别在于 Client 用户在文本字段中键入信息并按下回车键后才被动地将信息发送出去。当用户按下回车键后，将激活 action 方法（第 81 行）。用户在文本字段中键入的 String 将转换为一个 byte 数组，并且这个 byte 数组用来创建一个 DatagramPacket，DatagramPacket 使用 byte 数组、用户键入的 String 的长度、想要发送的目的地的地址（在本例中为 InetAddress.getLocalHost() 的返回值）以及 Server 等待接收的端口号来进行初始化，然后将数据报发送出去。如果发送时出现错误，则将抛出一个 IOException 异常。

Client 的 waitForPackets 方法通过下列语句的无限循环来等待接收数据报：

```
receiveSocket.receive( receivePacket );
```

这条语句直到接收了一个数据报时才结束执行。注意，这样做并不会妨碍用户发送数据报，仅仅是为了在一个数据报到达 Client 之前，while 循环能够一直执行。

每当一个数据报到达时，将其存储在 receivePacket 中，并将它的内容显示在文本字段内。如果接收一个数据报时产生了一个错误，系统将抛出一个 IOException 异常。用户可以在任何时刻将信息输入 Client 窗口的文本字段中并按下回车键，甚至是在接收一个数据报的时候。action 方法处理文本字段中的事件，并将发送包含文本字段中信息的数据报。

注意，本例中的客户必须了解服务器是在端口 5000 接收数据，而客户接收回应的端口号为 5001。

## 16.8 采用多线程服务器实现的客户/服务器间的三连棋游戏

在本节中,我们将讲述最重要的网络练习——通过采用流套接字技术的客户/服务器结构,实现一个大家常玩的三连棋游戏。这个程序由一个TicTacToeServer应用程序(如图16.7所示)组成。

```
1 // Fig. 16.7: TicTacToeServer.java
2 // This class maintains a game of Tic-Tac-Toe for two
3 // client applets.
4 import java.awt.*;
5 import java.net.*;
6 import java.io.*;
7
8 public class TicTacToeServer extends Frame {
9     private byte board[];
10    private boolean xMove;
11    private TextArea output;
12    private Player players[];
13    private ServerSocket server;
14    private int numberOfPlayers;
15    private int currentPlayer;
16
17    public TicTacToeServer()
18    {
19        super( "Tic-Tac-Toe Server" );
20        board = new byte[ 9 ];
21        xMove = true;
22        players = new Player[ 2 ];
23        currentPlayer = 0;
24
25        // set up ServerSocket
26        try {
27            server = new ServerSocket( 5000, 2 );
28        }
29        catch( IOException e ) {
30            e.printStackTrace();
31            System.exit( 1 );
32        }
33
34        output = new TextArea();
35        add( "Center", output );
36        resize( 300, 300 );
37        show();
38    }
39
40    // wait for two connections so game can be played
41    public void execute()
42    {
43        for ( int i = 0; i < players.length; i++ ) {
44            try {
45                players[ i ] =
46                    new Player( server.accept(), this, i );
47                players[ i ].start();
48                ++numberOfPlayers;
```

```
49         }
50         catch( IOException e ) {
51             e.printStackTrace();
52             System.exit( 1 );
53         }
54     }
55 }
56
57 public int getNumberOfPlayers() { return numberOfPlayers; }
58
59 public void display( String s )
60 {
61     output.appendText( s + "\n" );
62 }
63
64 // Determine if a move is valid.
65 // This method is synchronized because only one move can be
66 // made at a time.
67 public synchronized boolean validMove( int loc, int player )
68 {
69     boolean moveDone = false;
70
71     while ( player != currentPlayer ) {
72         try {
73             wait();
74         }
75         catch( InterruptedException e ) {
76         }
77     }
78
79     if ( !isOccupied( loc ) ) {
80         board[ loc ] =
81             (byte)( currentPlayer == 0 ? 'X' : 'O' );
82         currentPlayer = ++currentPlayer % 2;
83         players[ currentPlayer ].otherPlayerMoved( loc );
84         notify(); // tell waiting player to continue
85         return true;
86     }
87     else
88         return false;
89 }
90
91 public boolean isOccupied( int loc )
92 {
93     if ( board[ loc ] == 'X' || board[ loc ] == 'O' )
94         return true;
95     else
96         return false;
97 }
98
99 public boolean gameOver()
100 {
101     return false;
102 }
103
104 public boolean handleEvent( Event event )
```

```
105     {
106         if ( event.id == Event.WINDOW_DESTROY ) {
107             hide();
108             dispose();
109
110             for ( int i = 0; i < players.length; i++ )
111                 players[ i ].stop();
112
113             System.exit( 0 );
114         }
115
116         return super.handleEvent( event );
117     }
118
119     public static void main( String args[] )
120     {
121         TicTacToeServer game = new TicTacToeServer();
122
123         game.execute();
124     }
125 }
126
127 // Player class to manage each Player as a thread
128 class Player extends Thread {
129     Socket connection;
130     DataInputStream input;
131     DataOutputStream output;
132     TicTacToeServer control;
133     int number;
134     char mark;
135
136     public Player( Socket s, TicTacToeServer t, int num )
137     {
138         mark = ( num == 0 ? 'X' : 'O' );
139
140         connection = s;
141
142         try {
143             input = new DataInputStream(
144                 connection.getInputStream() );
145             output = new DataOutputStream(
146                 connection.getOutputStream() );
147         }
148         catch( IOException e ) {
149             e.printStackTrace();
150             System.exit( 1 );
151         }
152
153         control = t;
154         number = num;
155     }
156
157     public void otherPlayerMoved( int loc )
158     {
159         try {
160             output.writeUTF( "Opponent moved" );
```

```
161         output.writeInt( loc );
162     }
163     catch ( IOException e ) {}
164 }
165
166 public void run()
167 {
168     boolean done = false;
169
170     try {
171         control.display( "Player " +
172             ( number == 0 ? 'X' : 'O' ) + " connected" );
173         output.writeChar( mark );
174         output.writeUTF( "Player " +
175             ( number == 0 ? "X connected\n" :
176                 "O connected, please wait\n" ) );
177
178         // wait for another player to arrive
179         if ( control.getNumberOfPlayers() < 2 ) {
180             output.writeUTF( "Waiting for another player" );
181
182             while ( control.getNumberOfPlayers() < 2 )
183                 ;
184
185             output.writeUTF(
186                 "Other player connected. Your move." );
187         }
188
189         // Play game
190         while ( !done ) {
191             int location = input.readInt();
192
193             if ( control.validMove( location, number ) ) {
194                 control.display( "loc: " + location );
195                 output.writeUTF( "Valid move." );
196             }
197             else
198                 output.writeUTF( "Invalid move, try again" );
199
200             if ( control.gameOver() )
201                 done = true;
202         }
203
204         connection.close();
205     }
206     catch( IOException e ) {
207         e.printStackTrace();
208         System.exit( 1 );
209     }
210 }
211 }
```





图 16.7 采用客户/服务器结构实现的三连棋游戏程序的服务器端

TicTacToeServer 应用程序允许 2 个 TicTacToeClientApplet (如图 16.8 所示) 连接到服务器, 并且相互之间进行这个三连棋游戏 (输出如图 16.9 所示)。每当服务器接收到一个客户的连接申请时, 它就创建一个 Player 对象, 在一个单独的线程中处理这个客户的有关操作。这使得客户可以独立操作自己的游戏。首先连接上的客户将自动设定为 X (X 代表先走), 而第二个连接上的则设为 O。服务器维护有关棋盘的信息, 这样它可以判断一个客户的移动请求是否合法。每一个 TicTacToeClientApplet 维护自己的一个游戏棋盘, 在上面可以观察到游戏的进展情况。客户只能在棋盘空白的地方做上标志, Square 类用来实现棋盘上的 9 个空格。TicTacToeServer 类和 Player 类在文件 TicTacToeServer.java 中实现, TicTacToeClient 类和 Square 类在文件 TicTacToeClient.java 中实现。

```

1      // Fig. 16.8: TicTacToeClient.java
2      // Client for the TicTacToe program
3      import java.applet.Applet;
4      import java.awt.*;
5      import java.net.*;
6      import java.io.*;
7
8      // Client class to let a user play Tic-Tac-Toe with
9      // another user across a network.
10     public class TicTacToeClient extends Applet
11         implements Runnable {
12         TextField id;
13         TextArea display;
14         Panel boardPanel, panel2;
15         Square board[ ][ ], currentSquare;
16         Socket connection;
17         DataInputStream input;
18         DataOutputStream output;
19         Thread outputThread;
20         char myMark;
21
22         // Set up user-interface and board
23         public void init()
24         {
25             setLayout( new BorderLayout() );
26             display = new TextArea( 4, 30 );

```



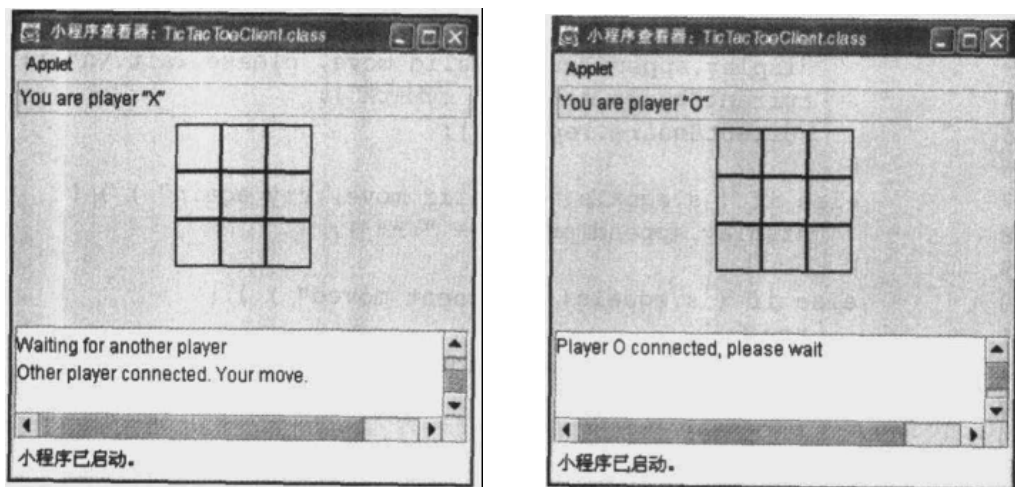
```
83         output.writeInt( row * 3 + col );
84     }
85 }
86     catch ( IOException ie ) {
87         ie.printStackTrace();
88     }
89 }
90 }
91     return true;
92 }
93
94 // Control thread that allows continuous update of the
95 // text area display.
96 public void run()
97 {
98     // First get player's mark (X or O)
99     try {
100         myMark = input.readChar();
101         id.setText( "You are player \"\" + myMark + \"\" );
102     }
103     catch ( IOException e ) {
104         e.printStackTrace();
105     }
106
107     // Receive messages sent to client
108     while ( true ) {
109         try {
110             String s = input.readUTF();
111             processMessage( s );
112         }
113         catch ( IOException e ) {
114             e.printStackTrace();
115         }
116     }
117 }
118
119 // Process messages sent to client
120 public void processMessage( String s )
121 {
122     if ( s.equals( "Valid move." ) ) {
123         display.appendText( "Valid move, please wait.\n" );
124         currentSquare.setMark( myMark );
125         currentSquare.repaint();
126     }
127     else if ( s.equals( "Invalid move, try again" ) ) {
128         display.appendText( s + "\n" );
129     }
130     else if ( s.equals( "Opponent moved" ) ) {
131         try {
132             int loc = input.readInt();
133
134             done:
135             for ( int row = 0; row < board.length; row++ )
136                 for ( int col = 0;
137                     col < board[ row ].length; col++ )
138                     if ( row * 3 + col == loc ) {
```

```

139         board[ row ][ col ].setMark(
140             ( myMark == 'X' ? 'O' : 'X' ) );
141         board[ row ][ col ].repaint();
142         break done;
143     }
144
145     display.appendText(
146         "Opponent moved. Your turn.\n" );
147     }
148     catch ( IOException e ) {
149         e.printStackTrace();
150     }
151 }
152 else {
153     display.appendText( s + "\n" );
154 }
155 }
156 }
157
158 // Maintains one square on the board
159 class Square extends Canvas {
160     char mark;
161
162     public Square()
163     {
164         resize ( 30, 30 );
165     }
166
167     public void setMark( char c ) { mark = c; }
168
169     public void paint( Graphics g )
170     {
171         g.drawRect( 0, 0, 29, 29 );
172         g.drawString( String.valueOf( mark ), 11, 20 );
173     }
174 }

```

图 16.8 采用客户/服务器结构实现的三连棋游戏程序的客户端



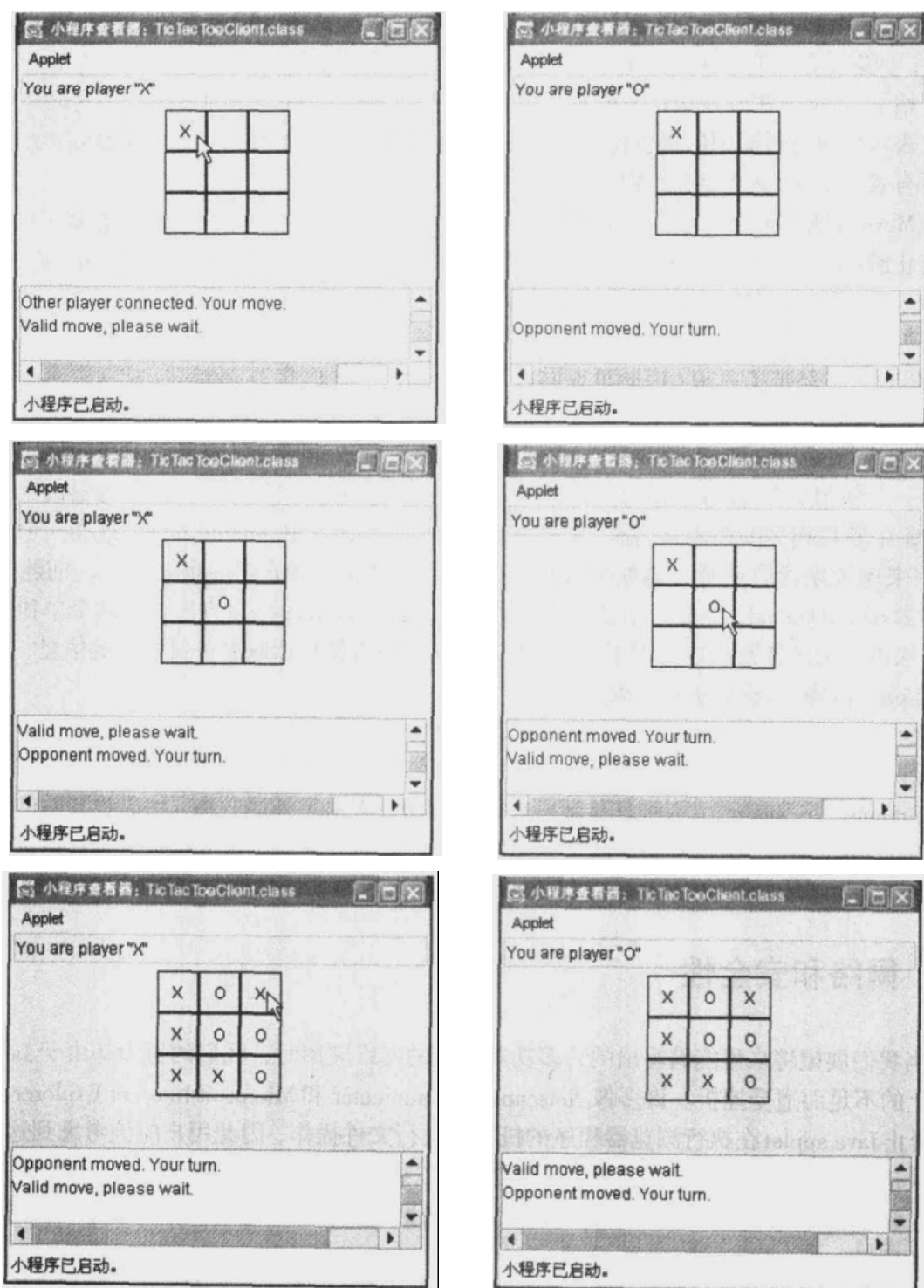


图 16.9 采用客户/服务器结构实现的三连棋游戏程序的输出范例

在开始执行 TicTacToeServer 应用程序之后，main 方法创建一个 TicTacToeServer 对象来调用游戏。构造函数试图设置一个 ServerSocket，如果成功，将显示服务器窗口并激活 TicTacToeServer 的 execute 方法。当接收到一个连接请求之后，程序将创建一个新的 Player 对象作为一个单独的线程来对这一连接进行管理，同时激活这个对象的 start 方法。

当创建 Player 时，它的构造函数需要代表连接到客户的 Socket 对象，并获得同输入和输出相关联的流。Player 的 run 方法控制发送到客户以及从客户接收的信息。首先，run 方法告诉客户连接已

经建立完；接着，每当客户在棋盘上走一步，该方法就将客户放在棋盘上的字符传递给另一个客户。其次，如果当前不存在两个客户，run 方法将一直循环，直到另一个客户登录上为止。这时游戏可以开始了，run 方法开始执行它的第二个 while 结构。这个 while 结构每执行一次循环，都会读入一个代表客户想进行标识的棋盘位置的整数，并且激活 TicTacToeServer 的 validMove 方法来检查移动是否有效。如果无效，将给客户发送一条信息。

validMove 方法将标识为 synchronized (同步)，这是为了限制客户每一时刻只能做一次移动。这样可以防止游戏双方同时修改有关棋盘的状态信息。如果 Player 试图走不属于自己的一步（即现在轮到对方走），则将该 Player 置为等待状态直到轮到他走。如果某一步试图走到已经标识的格子中，则向用户返回一个 false 结果。否则，服务器在用户自己的棋盘上标识出这一步的状态，通知另一个 Player 对象对方已经走了一步（因此可以向这个客户发送一条信息），激活 notify 方法使得正在等待的 Player（如果有）能够确认一步，并将 true 结果返回给该客户以表明这一步是有效的。

当开始执行一个 TicTacToeClient applet 时，程序将创建一个显示从服务器发送的信息的文本字段，以及一个使用 9 个 Square 对象的棋盘。这个 applet 的 start 方法打开一个同服务器的连接，并从 Socket 对象中获得相关的输入输出流。TicTacToeClient 类实现了 Runnable 界面，这样一个单独的线程可以用来连续地读取从服务器发往客户的信息。在同服务器的连接建立以后，创建并初始化 Thread 对象 outputThread，然后激活这一线程的 start 方法。applet 的 run 方法控制这个单独线程的执行。该方法首先从服务器读取标识字符（X 或 O），然后一直循环读取服务器发送的信息，这些信息将传递给 processMessage 方法进行处理。

如果收到的信息是字符串“Valid move.”，那么将显示信息“Valid move, please wait.”。同时将客户的标识设置到相应的空格中（用户点击的空格），并且重画这一空格。如果收到的信息是字符串“Invalid move, try again.”，则将其显示出来以提醒用户点击另外的空格。如果收到的信息是字符串“Opponent moved.”，则将从服务器那里读取一个代表对方移动位置的整数，并标识棋盘上相应的格子。如果收到任何其他的信息，仅仅将它们显示出来而不进行任何动作。

## 16.9 网络 and 安全性

正当我们展望将来可能编写出的许多功能强大的网络应用时，我们的努力却由于 Java 在网络安全性上的不足而遭受挫折。许多像 Netscape Communicator 和 Microsoft Internet Explorer 这样的浏览器，禁止 Java applet 在执行浏览器程序的机器上进行文件操作，因此用户应该考虑到这一点。通过可以从任意一台服务器上下载的 HTML 文档，可以把设计的 Java applet 发送到浏览器。通常，用户必须了解将在系统上执行的 Java applet 源代码，如果允许它们随意地操作文件系统，则可能会导致严重的后果。

另一种特殊的情况是要限制 applet 在执行时可以连接的机器。为了建立真正相互协作的应用程序，我们在理想状态下希望 applet 可以同任何地点的计算机进行通信，但是 Web 浏览器通常限制 applet，使得它仅仅可以同它的源服务器进行通信。

也许这些约束看起来太严格了。Java 是一个崭新的编程语言，它需要时间来发展成熟。我们需要有关 Java applet 和应用程序的更多实践经验，Java 的开发者需要时间来原因由于 Java 特殊体系结构和环境所导致的一些问题。

## 小结

- Java 的网络功能都包含在 java.net 这个软件包中。
- 流套接字提供面向连接的服务, 这种传输协议就是通常所说的 TCP (即传输控制协议)。
- UDP (即用户数据报协议)——是一种无连接的协议, 也就是在任何情况下它不保证数据报能送到目的地。事实上, 数据报可能丢失, 或者被复制, 甚至顺序混乱。
- HTTP 协议 (超文本传输协议) 使用 URL (即统一资源定位器) 在 Internet 上进行数据定位。
- URL 的构造函数判断传给它的 String 是否代表一个合法的 URL 地址。如果是, 则这个 URL 对象将初始化成包含这个统一资源定位器; 否则将抛出一个 MalformedURLException 异常。
- 使用 getAppletContext 方法 (从 Applet 类中继承) 得到一个 AppletContext 对象的引用, 这个对象代表了此 applet 的环境——即执行这个 applet 的浏览器。
- AppletContext 的 showDocument 方法将一个 URL 的对象作为参数接收, 然后在浏览器上显示与这个 URL 相关的 WWW 资源。
- URL 对象可以使用 openStream 方法取得一个同服务器上的与文件相关联的 InputStream 对象。
- ServerSocket 对象建立一个服务器, 用来等待客户连接请求的端口号, ServerSocket 构造函数的第二个参数设定了服务器能够等待请求及处理的客户数目。如果这个客户队列已经满了, 那么将自动拒绝来自客户的连接请求。
- 基于流的连接将由 Socket 对象处理。
- 使用 ServerSocket 的 accept 方法来等待一个连接。这个方法直到接收到一个连接时才能够终止。一旦接收了连接, 处理这个连接的 Socket 对象将赋给 connection。
- 通常, 可以使用 getInputStream 方法和 getOutputStream 方法来分别得到与 Socket 相关的 InputStream 和 OutputStream。
- 当信息传输完成后, 服务器通过 Socket 的 close 方法关闭连接。
- 当初始化一个套接字对象时, 通过指定服务器的名字和端口号来连接一个客户和服务。如果连接失败, 则将抛出一个 IOException 异常。
- UnknownHostException 是 IOException 的一个子类。
- 当 InputStream 的 read 方法读到流结束符时将返回 -1。
- 如果使用 DataInputStream 来从服务器读入信息, 当用户在读到流结束符之后还想从流中读出信息时 (使用不同于 readLine 的方法), 则将抛出一个 EOFException 异常。
- InetAddress 的 getLocalHost 方法返回一个程序正在其上执行的包含本地主机名的 InetAddress 对象。
- 客户同服务器连接的端口号有时也称为握手点。
- 面向连接的传输更像我们使用的电话系统: 拨打一个号码并同想要通话一方的电话机建立连接; 在使用的过程中这个连接将一直保持, 哪怕用户并没有在说话。
- 采用数据报的无连接传输方式更像通过邮局进行邮件传递。如果一份邮件太大以至于不能放在一个信封中, 只有将它分成几部分然后把它们分别放入单独编号的信封中, 每次寄出一封信。这些信件可能按照编号的顺序到达, 也可能不按顺序到达, 有的信件甚至会丢失。
- DatagramPacket 用来创建发送和接收信息的数据报。
- DatagramSocket 用来发送和接收 DatagramPacket。
- 不带参数的 DatagramSocket 构造函数用来设置在网络上发送数据报的套接字, 计算机将为

sendSocket 对象自动分配一个可用的端口号。

- 以一个整数端口号作为参数的DatagramSocket构造函数用来设置从网络上接收数据报的端口。
- 创建一个DatagramSocket时,指定一个正在使用的端口号或者指定一个不合法的端口号都将导致一个SocketException异常。
- DatagramSocket的receive方法将一直等待,直到一个数据报到达并将此数据报置于其参数中。
- DatagramPacket 的 getAddress 方法返回一个包含这个数据报源主机名的 InetAddress 对象。
- DatagramPacket 的 getPort 方法返回这个数据报通过端口进行传输的整数端口值。
- DatagramPacket 的 getLength 方法返回一个整数,该整数代表发送的数据按字节计算的长度。
- DatagramPacket 的 getData 方法返回包含发送的数据的字节数组。
- DatagramPacket 的构造函数包含4个参数:第一个参数指定byte这个数组将被发送,第二个参数指定发送的字节数,第三个参数指定这个数据报将发送到的客户方地址,第四个参数指定客户方在哪个端口等待接收数据报。
- DatagramSocket 的 send 方法在网络上发送一个 DatagramPacket。
- 无论是在接收或发送一个 DatagramPacket 时出现错误,系统都将抛出一个IOException异常。
- 许多像 Netscape Navigator 这样的 Web 浏览器禁止 Java applet 在它们执行的机器上进行诸如文件操作这样的动作。
- Web 浏览器通常限制一个 applet,使得它仅仅可以同其源服务器进行通信。

## 术语

accept a connection	接收一个连接	connectionless service	无连接服务
accept method of ServerSocket class	ServerSocket 类的 accept 方法	connectionless transmission with datagrams	采用数据报的无连接的传输方式
AppletContext		connection-oriented service	面向连接的服务
client	客户	connection request	连接请求
client connects to a server	客户同服务器连接	datagram	数据报
client-side socket	客户方套接字	DatagramPacket class	DatagramPacket 类
client-server relationship	客户/服务器的关系	datagram socket	数据报套接字
collaborative computing	协作计算	DatagramSocket class	DatagramSocket 类
close a connection	关闭一个连接	DataInputStream	
close method	close 方法	deny a connection	拒绝一个连接
close a Socket	关闭一个 Socket	duplicated packets	复制的数据包
close method of DataInputStream	DataInputStream 的 close 方法	getAddress method of DatagramPacket	DatagramPacket 类的 getAddress 方法
computer networking	计算机联网	getAppletContext	
connect method	connect 方法	getData method of class DatagramPacket	DatagramPacket 类的 getData 方法
connect to a port	连接到一个端口	getInputStream method of class Socket	Socket 类的 getInputStream 方法
connect to a World Wide Web site	连接到 WWW 网点	getLength method of DatagramSocket	
connection	连接		



- DatagramSocket 类的 getLength 方法
- getLocalHost method of DatagramPacket
- DatagramPacket 类的 getLocalHost 方法
- getOutputStream method of class Socket Socket 类的 getOutputStream 方法
- getPort method of class DatagramPacket
- DatagramPacket 类的 getPort 方法
- handshake point 握手点
- host 宿主
- import java.net 引入 java.net
- InputStream
- Internet
- Internet address Internet 地址
- InetAddress class InetAddress 类
- InetAddress.getLocalHost()
- IOException
- java.net package java.net 软件包
- Java RMI (Remote Method Invocation) API Java RMI (远程方法调用) API
- Java Server API
- Java Security API
- java.sun.com
- lost packages 丢失信息包
- MalformedURLException
- multithreaded server 多线程服务器
- Netscape Navigator
- network programming 网络编程
- networking 联网
- open a socket 打开一个套接字
- open a URL connection 打开一个 URL 连接
- openStream method of class URL URL 类的 openStream 方法
- out-of-sequence packets 顺序混乱的数据包
- OutputStream
- packet 数据包
- packet length 数据包长度
- port 端口
- port number on a server 服务器上的端口号
- read from a socket 从一个套接字中读取
- readline method of DataInputStream
- DataInputStream 的 readLine 方法
- receive method of class DatagramSocket
- DatagramSocket 类的 receive 方法
- register an available port number 登记一个合法的端口号
- send method of class DatagramSocket
- DatagramSocket 类的 send 方法
- server 服务器
- server-side socket 服务器一方的套接字
- ServerSocket class ServerSocket 类
- showDocument method of AppletContext
- AppletContext 的 showDocument 方法
- socket 套接字
- socket-based communications 基于套接字的通信
- Socket class Socket 类
- SocketException
- stream socket 流套接字
- TCP(Transmission Control Protocol) TCP(传输控制协议)
- UDP(User Datagram Protocol) UDP(用户数据报协议)
- UnknownHostException
- URL(Uniform Resource Locator) URL(统一资源定位器)
- URL class URL 类
- wait for a connection 等待一个连接
- wait for a packet 等待一个数据包
- Web browser Web 浏览器
- Web server Web 服务器
- World Wide Web 万维网 (WWW)

## 自测练习

### 16.1 填空:

- a) 在关闭一个套接字时, 如果出现一个 I/O 错误, 则将抛出一个 \_\_\_\_\_ 异常。

- b) 如果客户不能解析一个服务器的地址, 则将抛出 \_\_\_\_\_ 异常。
  - c) 如果一个 DatagramSocket 的构造函数不能正确设置它的对象, 则将抛出 \_\_\_\_\_ 异常。
  - d) URL 的构造函数将判断作为参数传递给它的 String 是否为合法的统一资源定位器。如果是, 那么这个 URL 对象将初始化为包含这个统一资源定位器; 否则将产生一个 \_\_\_\_\_ 异常。
  - e) Java 的有关网络的类都包含在 \_\_\_\_\_ 软件包中。
  - f) 对于不可靠的数据报传输, 应使用 \_\_\_\_\_ 类来创建一个套接字。
  - g) 一个 \_\_\_\_\_ 类的对象包含一个 Internet 的地址。
  - h) 本章讨论了 \_\_\_\_\_ 和 \_\_\_\_\_ 两种类型的套接字。
  - i) URL 这一缩写代表 \_\_\_\_\_。
  - j) 构成 WWW 的关键协议是 \_\_\_\_\_。
  - k) AppletContext 的 \_\_\_\_\_ 方法将一个 URL 对象作为参数而接收, 并且在浏览器中显示同这一 URL 相关联的 WWW 资源。
  - l) InetAddress 的 getLocalHost 方法返回一个正在运行本程序的、包含宿主主机名的 \_\_\_\_\_ 对象。
- 16.2 判断下述句子是否正确。如果不正确, 请解释原因。
- a) 一个 URL 对象创建后便不能改变。
  - b) UDP 是一种面向连接的协议。
  - c) 一个进程可以采用流套接字同另一个进程建立连接。
  - d) 服务器在每一端口上等待客户的连接请求。
  - e) 数据报的传输是可靠的, 可以保证数据包有序地到达。
  - f) 出于安全性考虑, 许多像 Netscape Navigator 这样的 Web 浏览器仅仅允许 Java applet 在执行它们的机器上进行文件处理。
  - g) Web 浏览器常常约束一个 applet, 使其仅能同下载它的机器进行通信。

## 自测练习答案

- 16.1 a) IOException。b) UnknownHostException。c) SocketException。d) MalformedURLException。e) java.net。f) DatagramSocket。g) InetAddress。h) 流, 数据报。i) 统一资源定位器。j) HTTP。k) showDocument。l) InetAddress。
- 16.2 a) 不正确。
- b) 正确。UDP 是面向非连接的协议而 TCP 是面向连接的协议。
  - c) 正确。
  - d) 正确。
  - e) 不正确。数据包可能丢失或者到达时是无序的。
  - f) 不正确。大多数浏览器不允许 applet 在客户上进行文件操作。
  - g) 正确。

## 练习

- 16.3 区别面向连接的网络服务和面向非连接的网络服务。
- 16.4 在客户端上, 一个客户如何判断宿主主机的名字?
- 16.5 在什么情况下程序会抛出一个 `SocketException` 异常?
- 16.6 客户如何从服务器上读取一行文本?
- 16.7 描述客户如何通过一个 URL 连接来从服务器上读取文件。
- 16.8 描述客户如何同服务器建立连接。
- 16.9 描述服务器如何将数据发送给客户。
- 16.10 描述如何准备一台服务器, 使得它可以接收单个客户提出的基于流的连接请求。
- 16.11 描述如何准备一台服务器, 使得它可以接收多个客户的连接请求, 并且在并行的进程中处理每个客户的连接。
- 16.12 一台服务器如何在一个端口上监听连接请求?
- 16.13 可以在队列中等待服务器处理的、来自客户的连接请求的最大数目是由什么决定的?
- 16.14 正如文中提及的, 导致一个服务器拒绝来自客户的连接请求的原因是什么?
- 16.15 采用套接字的连接方式, 允许客户向服务器提交一个文件名。如果文件存在就把文件内容发送给客户, 否则指出文件不存在。
- 16.16 修改前面的练习, 使得客户可以修改文件的内容, 并将结果发还给服务器进行存储。用户可以在一个文本字段中编辑文件, 完成后点击一个 “save changes” 按钮将文件传输给服务器。
- 16.17 修改图 16.1 中的程序以便在一个 List 对象中显示一个网站列表。允许用户把他们自己的网站增加到列表中或从列表中删去网站。
- 16.18 由于具有多处理器的服务器的广泛使用, 当今的多线程服务器已经非常普遍。修改 16.6 节中的例程, 将其中的简单服务器改为多线程服务器, 然后同时使多个客户同这个服务器进行连接。
- 16.19 在本章中, 我们演示了一个由多线程服务器控制的三连棋游戏的例子。仿照三连棋游戏开发一个方格游戏, 两个用户交替地走棋, 在程序中判断轮到谁走并且走法是否合理。由用户自己判断游戏是否结束。
- 16.20 模仿上一个练习, 开发一个象棋游戏。
- 16.21 开发一个 21 点的扑克游戏, 其中由服务器给每个用户发牌, 服务器作为客户的分牌手 (同真实游戏中分牌手的角色一样)。
- 16.22 开发一个扑克游戏, 由服务器将纸牌分发给每个用户, 服务器将作为客户的分牌手 (同真实游戏中分牌手的角色一样)。
- 16.23 (对多线程的三连棋游戏的修改) 图 16.7 和图 16.8 中的程序实现了一个多线程客户/服务器版本的三连棋游戏, 我们的目标是将程序修改为在同一时刻可以处理来自客户的多个连接。示例中的服务器成为两个 applet 之间真正的裁判——由它判断该轮到谁走以及走法是否合理, 服务器不判断谁胜谁负或者有人弃权。同时, 不允许服务器在一个新游戏中扮演玩家的角色或者由服务器主动终止现有的游戏。下面是对程序进行修改的建议。
  - a) 修改 `TicTacToeServer` 类来测试游戏的赢、输或者弃权, 当游戏结束时向每个客户发送一个表明结果的信息。
  - b) 修改 `TicTacToeClient` 类来增加一个新的按钮, 客户点击它时可以启动一个新的游戏,

仅当一个游戏已经结束时才能激活这个按钮。注意, `TicTacToeClient` 类和 `TicTacToeServer` 类必须进行相应的修改, 以重置棋盘和所有的状态信息。同时, 应该通知另一个 `TicTacToeClient` 重新开始一个游戏, 同时也将重新设置它的棋盘和状态。

- c) 修改 `TicTacToeClient` 类, 提供一个可以让用户在任何时刻终止游戏的按钮, 并且在点击这个按钮后, 服务器和其他客户应该得到通知, 服务器应该等待另一个客户的重新连接以便再开始一个游戏。
- d) 修改 `TicTacToeClient` 类和 `TicTacToeServer` 类, 使得当前游戏的胜者可以在之后的游戏中选择 X 或者 O, X 代表先走。
- e) 如果有兴趣, 可以允许一个客户在服务器等待其他客户连接的同时和服务器下棋。

16.24 ( 三维的三连棋游戏 ) 修改现有的多线程、基于客户服务器的三连棋游戏, 使得它成为一个三维的、 $4 \times 4 \times 4$  个的版本。让服务器能够简单地监控两个客户。把一个三维棋盘显示为 4 个棋盘, 其中每个含 4 行 4 列。如果读者还有更大的兴趣, 不妨尝试下述修改:

- a) 按照三维视图的方式绘制棋盘;
- b) 允许服务器判断输赢或者弃权。注意, 在一个  $4 \times 4 \times 4$  的棋盘中存在许多种赢的可能性。

# 第17章 数据结构

## 教学目标

- 能够使用引用、自引用的类以及递归定义来构造链接的数据结构
- 能够创建并利用诸如链表、队列、堆栈和二叉树这样的动态数据结构
- 理解利用动态数据结构的各种不同的重要应用
- 理解如何创建采用类、继承关系和复合关系的可重用的数据结构

## 17.1 简介

我们已经学习了一些固定长度的数据结构,例如一维数组和二维数组。本章将介绍能够在执行时容量增大或缩小的动态数据结构。链表可将数据单元“连成一行”,可以在一个链表的任何位置上进行插入和删除操作。堆栈在编译器和操作系统中非常重要,仅能在堆栈的一端(我们称为栈顶)进行插入和删除操作。队列相当于人们进行等待时形成的排队,允许在队列的后面(通常称为队尾)进行插入操作,或在队列的前面(通常也称为队首)开始删除操作。二叉树能够实现对数据的高速查找和存储,可以有效地删除重复的数据单元,表示文件系统的目录结构,以及将表达式编译为机器码。这些数据结构还有许多有趣的应用领域。

本章将讨论数据结构的每一种主要类型,并且实现一些创建和利用这些数据结构的程序。出于重用和维护的目的,我们使用类及类的继承和复合来创建和封装这些数据结构。

本章的程序都非常实用,可以应用在更高级的项目和工业领域中,这些程序特别注重于对引用的利用,习题中包括大量有用的应用程序。

我们鼓励读者完成特殊小节“建立自己的编译器”中的主要项目。在实际应用中,我们使用一个编译器来将Java程序转换为8位代码集,从而在计算机上执行这些程序。在这个特殊的项目中,读者可以建立自己的一个真正的编译器。这个编译器可以识别采用一种简单而又功能强大、类似Basic早期版本的语言所编写的程序,然后将这种语言写出的程序转化为由Simpletron机器语言(SML)的指令组成的文件(我们在第5章的特殊小节“建立自己的计算机”中介绍了SML),最后就可以使用Simpletron模拟程序来执行编译器产生的SML程序。采用面向对象的编程概念来实现这样一个项目,将提供给读者一个运用在本书中学到的知识的大好机会。本章的特殊小节详细地向读者介绍了高级语言的定义,以及用来将高级语言的程序转化为机器语言的必要算法。如果读者喜欢一些富有挑战性的事情,不妨试试练习中有关编译器和Simpletron模拟程序的题目。

## 17.2 自引用的类

一个自引用的类包含这样一个数据成员,该成员指向与自身类型相同的一个对象。例如,以下语句定义了一个Node类:

```

class Node {
    private int data;
    private Node next;

    Node( int d ) { /* method body */
    void setData (int d ) { /* method body */
    int getData( ){ /* method body */
    void setNext( Node nextNode ) { /* method body */
    Node getNext( ) { /* method body */

```

Node 类有两个私有实例变量——整数 data 和 Node 的引用 next。成员 next 引用 Node 类型的一个对象——同这一成员类型相同的一个对象，因此我们称其为“自引用的类”。也可以将成员 next 看成指针——即 next 可以将一个 Node 类型的对象和另一个相同类型的对象相关联。Node 类有 5 个方法：一个用来初始化 data 的、接收整数参数的构造函数，setData 方法用来设置 data 的值，getData 方法可以返回 data 的值，setNext 方法设置 next 的值，getNext 方法返回成员 next 的值。

可以把自引用的对象连接在一起，从而构成许多有用的数据结构，例如链表、队列、堆栈和树。图 17.1 中描绘了两个自引用的对象如何连接在一起，从而构成一个链表。一条斜杠放置在第二个自引用对象的链接成员部分上，它代表了一个 null 的引用，表明这个链接不指向任何其他对象。这条斜杠仅仅用于描述的目的，这同 Java 语言中的反斜杠字符是不同的。一个 null 字符通常代表了一个数据结构的终止。

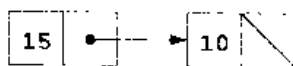


图 17.1 两个自引用的对象链接在一起

#### 常见编程错误 17.1

没有将一个链表最后一个节点的指针设置为 null。

## 17.3 动态内存请求

创建并维护动态数据结构需要动态内存请求——即程序在执行时要获得更多的内存空间来放置新节点或释放不再需要的空间。Java 程序不需要显式地说明需要释放的动态内存空间，实际上，Java 将自动完成这些内存无用单元的收集工作。动态内存请求的极限在机器上通常为—台机器的物理内存大小；或者在一个虚拟内存系统上，则为系统能够提供的最大虚拟内存的大小。通常这个极限是相当小的，因为计算机的内存一般要由许多用户所共享，所以一个用户的可支配空间就相当有限了。

运算符 new 用来请求内存。运算符 new 使用请求内存的对象的类型作为参数，而返回请求完成时这个类型对象的一个引用。例如，下列语句：

```
Node nodeToAdd = new Node( 10 );
```

为存储一个 Node 分配相应的空间，并将指向这块内存的引用赋给 nodeToAdd。如果请求不到内存，new 将抛出一个 OutOfMemoryException 异常，语句中的 10 是 Node 对象的值。

下面各节分别讨论了链表、堆栈、队列和树，这些数据结构采用了自引用的类和动态内存请求来进行创建和维护。

### 编程技巧 17.1

当使用 `new` 运算符时，应测试 `OutOfMemoryException` 异常。如果没有得到请求的内存，就需要进行相应的错误处理。

## 17.4 链表

链表是一些自引用对象的线性集合，这些对象称为节点，由于采用引用链将这些对象连接起来，因此称其为链表。可以通过链表第一个节点的引用而得到整个链表。通常，链表的最后一个节点的引用将设置为 `null`，以便标识这个链表结束。数据将动态地存储在一个链表中，只有在必要时才创建某个节点。节点中可以包含任何类型的数据，甚至是其他类的对象。堆栈和队列都是线性的数据结构，我们在后面将会看到，它们相当于具有各自约束的链表类型。树不是线性的数据结构。

一系列的数据可以放置在数组中，但是链表具有几个优点。在数据成员的数目事先不能得知的情况下，链表能够提供更适当的存储空间。链表是动态的，因此可以根据需要来增加或减少它的大小。一个“传统的”Java数组的大小是不可以修改的，这是因为在创建数组时就已经定义了大小。“传统的”数组可能会被数据占满，而链表仅仅在系统无法提供足够的内存来满足动态存储空间的请求时才会无法扩充。Java API 的 `java.util` 软件包中包含用来实现和维护动态数组的 `Vector` 类，它可以在程序执行时增加或缩小，我们将在第18章中介绍 `Vector` 类。

### 性能提示 17.1

可以在定义数组时声明比实际使用长度更长的数据成员，但是这样会浪费内存。在这种情况下，链表可以提供对内存的更有效使用，链表允许程序在执行时调整其大小。

### 性能提示 17.2

在一个链表中插入成员是很快速的——仅仅需要改变两个引用，所有已有的节点都保留在内存的原有位置。

可以通过将新节点插入在相应的位置来有序地保存链表，已有的成员不需要移动。

### 性能提示 17.3

在一个有序的数组中执行插入和删除命令会相当耗时——在插入和删除位置之后的所有成员都要适当地移动。

### 性能提示 17.4

数组的成员在内存中连续存放。这使得我们可以通过数组第一个元素的位置以及要访问的元素下标来直接计算出需要访问的元素的内存地址。链表不具有这种可以直接计算出元素地址的特性。

链表的节点通常并不连续存放在内存中。尽管这样，从逻辑上我们可以认为这些节点元素是连续的。图 17.2 中显示了一个具有好几个节点的链表。

### 性能提示 17.5

使用在程序执行过程中能够增加或缩小容量的动态数据结构可以节约内存。尽管这样，我们应当谨记链表中的引用也是要占用一定空间，并且动态内存请求将增加调用方法时的开销。

图 17.3（它的输出显示在图 17.4 中）的程序使用了一个 `List` 类来操纵链表中各种不同的对象类型。`ListTest` 类的 `main` 方法创建了一个对象链表，使用 `insertAtFront` 方法在链表的首部插入对象，使用 `insertAtBack` 方法在链表的尾部插入对象，使用 `removeFromFront` 方法从链表的首部删除对象，

使用 `removeFromBack` 方法从链表的尾部删除对象。在每一个插入和删除操作结束之后，激活 `print` 方法并显示链表的内容。下面将对这个程序进行详细讨论。如果试图从一个空链表中删除节点，则程序将抛出一个 `EmptyListException` 异常（如图 17.3 的第 140 行定义的）。练习 17.20 要求读者实现一个递归的方法，从后向前显示链表中的节点内容；练习 17.21 中要求读者实现一个递归的方法，可以在一个链表中搜寻某一特定的数据项。

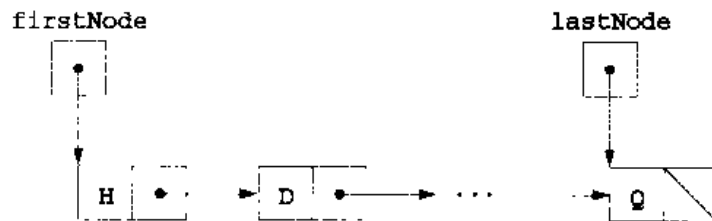


图 17.2 一个链表的图示

```

1  // Class ListNode definition
2  class ListNode {
3      // friendly data so class List can access it directly
4      Object data;
5      ListNode next;
6
7      // Constructor: Create a ListNode that refers to Object o.
8      ListNode( Object o )
9      {
10         data = o;      // this node refers to Object o
11         next = null;   // set next to null
12     }
13
14     // Constructor: Create a ListNode that refers to Object o and
15     // to the next ListNode in the List.
16     ListNode( Object o, ListNode nextNode )
17     {
18         data = o;      // this node refers to Object o
19         next = nextNode; // set next to refer to next
20     }
21
22     // Return the Object in this node
23     Object getObject() { return data; }
24
25     // Return the next node
26     ListNode getNext() { return next; }
27 }
28
29 // Class List definition
30 class List {
31     private ListNode firstNode;
32     private ListNode lastNode;
33     private String name; // String like "list" used in printing
34
35     // Constructor: Construct an empty List with s as the name
36     public List( String s )
37     {
38         name = s;
39         firstNode = lastNode = null;

```



```
40     }
41
42     // Constructor: Construct an empty List with
43     // "list" as the name
44     public List() { this( "list" ); }
45
46     // Insert an Object at the front of the List
47     // If List is empty, firstNode and lastNode refer to
48     // same Object. Otherwise, firstNode refers to new node.
49     public void insertAtFront( Object insertItem )
50     {
51         if ( isEmpty() )
52             firstNode = lastNode = new ListNode( insertItem );
53         else
54             firstNode = new ListNode( insertItem, firstNode );
55     }
56
57     // Insert an Object at the end of the List
58     // If List is empty, firstNode and lastNode refer to
59     // same Object. Otherwise, lastNode's next instance
60     // variable refers to new node.
61     public void insertAtBack( Object insertItem )
62     {
63         if ( isEmpty() )
64             firstNode = lastNode = new ListNode( insertItem );
65         else
66             lastNode = lastNode.next = new ListNode( insertItem );
67     }
68
69     // Remove the first node from the List.
70     public Object removeFromFront() throws EmptyListException
71     {
72         Object removeItem = null;
73
74         if ( isEmpty() )
75             throw new EmptyListException( name );
76
77         removeItem = firstNode.data; // retrieve the data
78
79         // reset the firstNode and lastNode references
80         if ( firstNode.equals( lastNode ) )
81             firstNode = lastNode = null;
82         else
83             firstNode = firstNode.next;
84
85         return removeItem;
86     }
87
88     // Remove the last node from the List.
89     public Object removeFromBack() throws EmptyListException
90     {
91         Object removeItem = null;
92
93         if ( isEmpty() )
94             throw new EmptyListException( name );
95
96         removeItem = lastNode.data; // retrieve the data
97     }
```

```

98         // reset the firstNode and lastNode references
99         if ( firstNode.equals( lastNode ) )
100             firstNode = lastNode = null;
101         else {
102             ListNode current = firstNode;
103
104             while ( current.next != lastNode )
105                 current = current.next;
106
107             lastNode = current;
108             current.next = null;
109         }
110
111         return removeItem;
112     }
113
114     // Return true if the List is empty
115     public boolean isEmpty() { return firstNode == null; }
116
117     // Output the List contents
118     public void print()
119     {
120         if ( isEmpty() ) {
121             System.out.println( "Empty " + name );
122             return;
123         }
124
125         System.out.print( "The " + name + " is: " );
126
127         ListNode current = firstNode;
128
129         while ( current != null ) {
130             System.out.print( current.data.toString() + " " );
131             current = current.next;
132         }
133
134         System.out.println();
135         System.out.println();
136     }
137
138
139     // Class EmptyListException definition
140     class EmptyListException extends RuntimeException {
141         public EmptyListException( String name )
142         {
143             super( "The " + name + " is empty" );
144         }
145     }
146
147     // Class ListTest
148     public class ListTest {
149         public static void main( String args[] )
150         {
151             List objList = new List(); // create the List container
152
153             // Create objects to store in the List
154             Boolean b = new Boolean( true );
155             Character c = new Character( 'S' );
156             Integer i = new Integer( 34567 );

```

```

156     String s = new String( "hello" );
157
158     // Use the List insert methods
159     objList.insertAtFront( b );
160     objList.print();
161     objList.insertAtFront( c );
162     objList.print();
163     objList.insertAtBack( z );
164     objList.print();
165     objList.insertAtBack( s );
166     objList.print();
167
168     // Use the List remove methods
169     Object removedObj;
170
171     try {
172         removedObj = objList.removeFromFront();
173         System.out.println( removedObj.toString() + " removed" );
174
175         objList.print();
176         removedObj = objList.removeFromFront();
177         System.out.println( removedObj.toString() + " removed" );
178
179         objList.print();
180         removedObj = objList.removeFromBack();
181         System.out.println( removedObj.toString() + " removed" );
182
183         objList.print();
184         removedObj = objList.removeFromBack();
185         System.out.println( removedObj.toString() + " removed" );
186
187         objList.print();
188     }
189     catch ( EmptyListException e ) {
190         System.err.println( " \n" + e.toString() );
191     }
192 }
193 }

```

图 17.3 操纵一个链表

```

The list is: true
The list is: $ true
The list is: true 34567
The list is: true 34567 hello
$ removed
The list is: true 34567 hello
hello removed
The list is: 34567
34567 removed
Empty list

```

图 17.4 图 17.3 中程序的输出示例

图 17.3 的程序中包括三个类——`ListNode`、`List` 和 `EmptyListException`，封装在每个 `List` 对象中的是一个 `ListNode` 的对象链表。`ListNode` 类包括两个成员：`data` 和 `next`，`ListNode` 的成员 `data` 可以指向任何一个 `Object`，`ListNode` 的成员 `next` 将链表中下一个 `ListNode` 对象的引用存放在 `next` 中。

`List` 类包含私有成员 `firstNode`（指向一个链表首部的 `ListNode` 的引用）和 `lastNode`（指向一个链表结尾的 `ListNode` 的引用）。默认的构造函数将两个引用在默认情况下初始化为空。`List` 类的主要方法为 `insertAtFront`、`insertAtBack`、`removeFromFront` 以及 `removeFromBack`。`isEmpty` 方法称为谓词方法——它不会对链表进行任何修改；尽管这样，该方法仍可以用来判定链表是否为空（即指向链表第一个节点的引用为 `null`）。如果链表为空，则返回 `true`；否则返回 `false`。`print` 方法用来显示链表的内容。

下面我们将详细讨论 `List` 类中的每一个方法，`insertAtFront` 方法（图 17.5 描绘了这个操作）将一个节点放置在链表的首部。这个方法包含以下几个步骤：

1. 调用 `isEmpty` 方法判断链表是否为空（第 51 行）。
2. 如果链表为空，`firstNode` 和 `lastNode` 都将设置为指向 `new` 运算符产生的 `ListNode` 对象，并且利用 `insertItem`（第 52 行）进行初始化。`ListNode` 的构造函数将实例变量 `data` 的值设为作为参数传递进来的 `insertItem`，同时将 `next` 引用置为 `null`。
3. 如果链表不为空，则将一个新采用 `new` 运算符创建的，并使用 `insertItem` 和 `firstNode` 进行初始化的 `ListNode` 对象（第 54 行）加入链表中，同时将 `firstNode` 指向这一对象。每当执行 `ListNode` 的构造函数（第 16 行）时，它将实例变量 `data` 的值设置为作为参数传递进来的 `insertItem`，同时将实例变量 `next` 指向作为参数传递进来的 `ListNode` 对象。

图 17.5 描述了 `insertAtFront` 方法。图中的 a) 部分表明在新的节点还没有插入链表之前，程序执行 `insertAtFront` 操作时的链表和新节点。b) 部分中带点的箭头描绘了 `insertAtFront` 操作的第 3 步，我们看到包含数据 12 的节点成为新的表头。

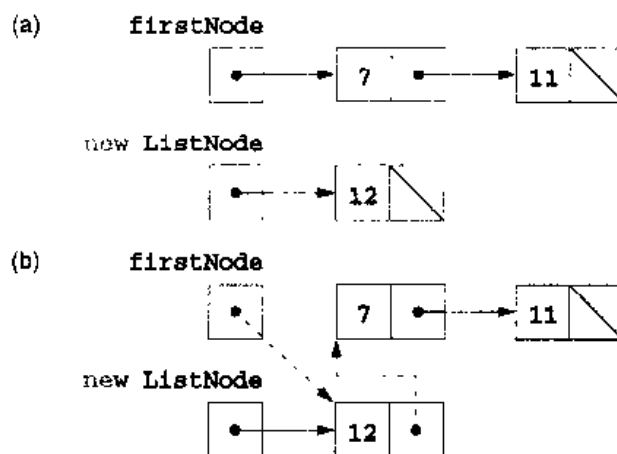


图 17.5 `insertAtFront` 的图形描述

`insertAtBack` 方法（图 17.6 描述了这一操作）将一个新的节点放在链表的尾部，这个方法包含以下几个步骤：

1. 调用 `isEmpty` 来判断链表是否为空（第 63 行）。
2. 如果链表为空，就通过 `new` 运算符为 `ListNode` 分配空间，并采用 `insertItem`（第 64 行）将其初始化，同时将 `firstNode` 和 `lastNode` 都指向 `ListNode`。`ListNode` 的构造函数（第 8 行）将实

- 例变量 `data` 的值设置为指向作为参数传递的 `insertItem`，同时将 `next` 引用设置为 `null`。
- 如果链表不为空，就采用 `new` 运算符来分配空间，并使用 `insertItem` 进行初始化，所产生的 `ListNode` 将作为新节点而加入链表中，我们将 `lastNode` 和 `lastNode.next` 指向这一新节点（第 66 行）。当执行 `ListNode` 的构造函数（第 8 行）时，它将实例变量 `data` 的值设置为指向作为参数传递的 `insertItem` 对象，同时将 `next` 引用设置为 `null`。

图 17.6 描绘了 `insertAtBack` 的操作。图中的 a) 部分表明了 `insertAtBack` 的操作过程中，新节点还没有加入链表前的链表和新节点的状态。b) 部分带点的箭头描绘了方法 `insertAtBack` 将一个节点加在一个不为空的链表尾部。

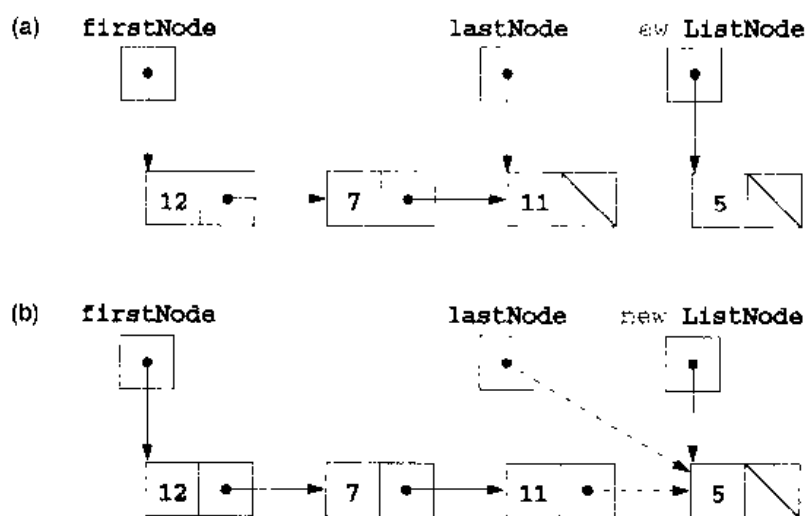
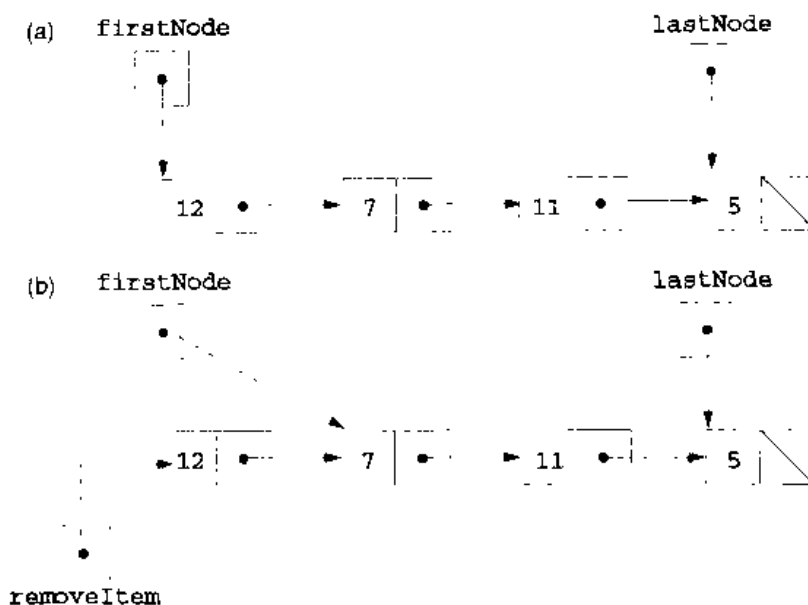


图 17.6 `insertAtBack` 操作的图形描述

`removeFromFront` 方法（在图 17.7 中描绘）将链表中第一个节点删除，同时返回删除的数据的引用。如果试图从空的链表中删除节点，则该方法将抛出一个 `EmptyListException` 异常（第 74 行 ~ 第 75 行），否则将返回删除的节点的引用。这个方法包含下面几个步骤：

1. 将 `removeItem` 设置为指向 `firstNode.data`（将要从链表中删除的数据）。
2. 如果 `firstNode` 等于 `lastNode`（第 80 行），即如果这个链表在执行删除操作之前仅有一个节点，我们就将 `firstNode` 和 `lastNode` 同时置为 `null`（第 81 行），这也就意味着将节点从链表中移去（将链表置为空）。
3. 如果在执行删除操作之前链表中有超过一个的节点数目，我们可以不管 `lastNode` 而简单地将 `firstNode` 指向 `firstNode.next`（第 83 行），即将 `firstNode` 指向我们在删除前的第二个节点（现在是第一个节点）。
4. 返回 `removeItem` 引用。

图 17.7 描绘了 `removeFromFront` 方法。图中的 a) 部分表示在执行删除操作前的链表，b) 部分表示具体的有关引用的修改。

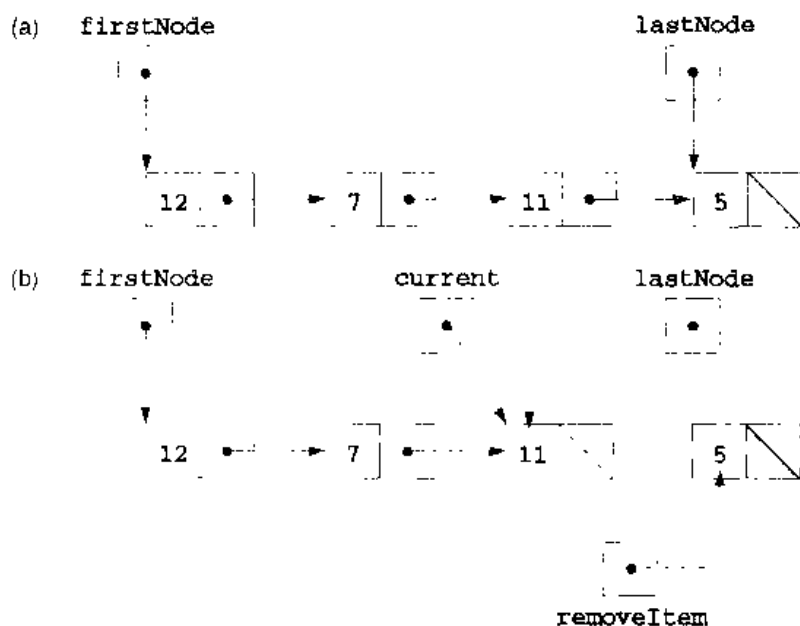
图 17.7 `removeFromFront` 操作的图形描述

`removeFromBack` 方法（在图 17.8 中描绘）将链表的最后一个节点删除，并返回删除的数据的引用。如果试图从一个空的链表中删除节点，那么 `removeFromBack` 方法将抛出一个 `EmptyListException` 异常（第 93 行~第 94 行），否则返回删除的数据的指针。这个方法包含以下几个步骤：

1. 将 `removeItem` 指向 `lastNode.data`（将从链表中删除的数据）。
2. 如果 `firstNode` 等于 `lastNode`（第 99 行），即如果这个链表在执行删除操作之前仅有一个节点，我们就将 `firstNode` 和 `lastNode` 同时置为 `null`（第 100 行），这也就意味着将节点从链表中删除（将链表置为空）。
3. 如果在执行删除操作之前链表中有超过一个的节点数目，那么就创建一个 `ListNode` 的引用 `current`，并将其初始化为 `firstNode`。
4. 现在使用遍历整个链表，直到它指向最后一个节点之前的那个节点。这个步骤通过一个 `while` 循环来不断地利用 `current.next` 来替代 `current`，直到 `current.next` 指向 `lastNode`。
5. 将 `lastNode` 指向 `current` 指向的对象，这意味着将最后的节点从链表中删除。
6. 将现在链表中最后一个节点的 `next` 引用，也就是 `current.next` 设置为 `null`。
7. 返回 `removeItem` 引用。

图 17.8 描绘了方法 `removeFromBack`。图中的 a) 部分说明了在执行删除操作之前的链表状态，b) 部分显示了实际的引用变化。

`print` 方法首先判断链表是否为空。如果是，则 `print` 显示 “The List is Empty” 并结束执行，否则它将显示链表中的数据。这个方法显示这样的字符串：其中包含字符 “The”，`String` 对象 `name`，以及字符串 “is:”。然后使用 `firstNode` 来创建和初始化 `ListNode` 的引用 `current`。循环显示 `current.data.toString()` 的结果，并将 `current.next` 的值赋给 `current`，直到 `current` 的值为 `null`。注意，如果链表中最后一个节点的指针不为空，则打印算法将由于遍历超过链表的尾部而导致错误。链表、堆栈和队列的打印算法是类似的。

图 17.8 `removeFromBack` 操作的图形表示

## 17.5 堆栈

堆栈是一个压缩版本的链表——仅能从一个堆栈的顶部添加或删除新的节点。正因为如此,我们称堆栈是一种后进先出 (LIFO) 的数据结构。堆栈的最后一个节点的链接成员将设置为 `null`, 用来表明堆栈的底部。

### 常见编程错误 17.2

忘记将一个堆栈底部节点的链接指针设置为 `null`。

操作一个堆栈的主要方法是 `push` (推入) 和 `pop` (弹出), `push` 方法在堆栈的顶部增加一个新的节点, `pop` 方法将堆栈顶部的节点删除。

堆栈可以生成许多有趣的应用。例如, 当调用一个方法时, 调用的方法必须知道怎样返回到调用它的位置, 因此返回地址将压入到堆栈中。如果连续调用一系列的方法, 这些连续的返回地址将按照后进先出的顺序压入堆栈, 这样每一个调用的方法就可以返回到调用者的地址。堆栈支持递归的方法调用, 该方法同传统的非递归的方法调用一样。

每当激活一个方法时, 堆栈包含创建其中的自动变量所需的空间。当一个方法返回到它的调用者之后, 这个方法的自动变量所占用的空间将从堆栈中弹出, 并且这些变量在程序中将不再可见。

堆栈在编译器分析表达式和产生机器码时大量使用。本章的习题涉及有关堆栈的一些应用, 包括使用它们来开发出一个完整的可工作的编译器。

我们将利用链表和堆栈之间的紧密关系, 并主要通过重用一个链表的类来实现一个堆栈的类。我们使用两种不同的重用方式。首先, 通过从 `List` 类继承来产生堆栈的类。然后, 通过将一个 `List` 对象作为它的一个 `private` 成员, 我们可以实现一个拥有同样功能的堆栈类。本章讨论的链表、堆栈和队列的数据都指定为 `Object` 的对象, 这样可以方便将来的重用。因此, 任何对象类型都可以存储在我们描述的链表、堆栈和队列中。

图 17.9 (输出在图 17.10 中) 通过继承图 17.3 中的 `List` 类来创建一个堆栈类。我们希望这个堆

栈拥有 push 方法、pop 方法、isEmpty 方法和 print 方法。这些方法本来是 List 类中的 insertAtFront 方法、removeFromFront 方法、isEmpty 方法和 print 方法。当然，List 类还包含其他的方法（即 insertAtBack 和 removeFromBack 方法），但是我们在实现堆栈类时将不再允许通过公有接口进行访问。有一点非常重要，List 类中所有的公有方法，在它的继承类 StackInheritance 中同样也是 public 类型的。我们在图 17.11 中演示了另外一种建立堆栈类的手段：当实现堆栈中的方法时，我们让每一个 StackInheritance 的方法调用 List 的方法——push 调用 insertAtFront、pop 调用 removeFromFront、isEmpty 调用 super.isEmpty，从而激活基类中的相应版本；print 也将调用 super.print 来激活基类中的版本。

```

1      // Class StackInheritance definition
2      // Derived from class List
3      class StackInheritance extends List {
4          public StackInheritance() { super( "stack" ); }
5          public void push( Object o ) { insertAtFront( o ); }
6          public Object pop() throws EmptyListException
7              { return removeFromFront(); }
8          public boolean isEmpty() { return super.isEmpty(); }
9          public void print() { super.print(); }
10     }
11     // Class StackInheritanceTest
12     public class StackInheritanceTest {
13         public static void main( String args[] )
14         {
15             StackInheritance objStack = new StackInheritance();
16
17             // Create objects to store in the stack
18             Boolean b = new Boolean( true );
19             Character c = new Character( 'S' );
20             Integer i = new Integer( 34567 );
21             String s = new String( "hello" );
22
23             // Use the push method
24             objStack.push( b );
25             objStack.print();
26             objStack.push( c );
27             objStack.print();
28             objStack.push( i );
29             objStack.print();
30             objStack.push( s );
31             objStack.print();
32
33             // Use the pop method
34             Object removedObj = null;
35
36             try {
37                 while ( true ) {
38                     removedObj = objStack.pop();
39                     System.out.println( removedObj.toString() +
40   " popped" );
41                     objStack.print();
42                 }
43             }
44             catch ( EmptyListException e ) {

```



---

```

45         System.err.println( " \n" + e.toString() );
46     }
47
48 }

```

---

图 17.9 一个简单的堆栈程序

---

```

The stack is: true
The stack is: $ true
The stack is: 34567 $ true
The stack is: hello $ true
hello popped
The stack is: $ true
$ popped
The stack is: true
true popped
Empty stack
EmptyListException: The stack is empty

```

---

图 17.10 图 17.9 中程序的输出结果

在 `StackInheritanceTest` 的 `main` 方法中，使用 `StackInheritance` 类来实例化一个称为 `objStack` 的 `Object` 堆栈。其中，一个 `Boolean` 对象包含 `true`，一个 `Character` 对象包含 `$`，一个 `Integer` 对象包含 `34567`，以及一个 `String` 对象包含 `hello`，它们都将推入堆栈 `objStack` 中并先后从中弹出。这些对象在一个无限循环的 `while` 中将逐步弹出。当堆栈中没有对象时，将抛出一个 `EmptyListException` 异常，并显示一条信息来说明堆栈为空。

另一个实现堆栈的方法是通过复合类来重用一個链表类。图 17.11 中的程序在 `StackComposition` 类的定义部分使用了一个私有的 `List` 对象（第 3 行）。类的复合允许我们将 `List` 类中的全部方法隐藏，而仅将其中需要的部分通过堆栈类的公有接口进行引用。这种通过调用相应的 `List` 方法来实现堆栈方法的技术称为 `forwarding`——堆栈的方法向前激活相应的 `List` 中的方法。`StackCompositiond-Test` 类使用一个和 `StackInheritanceTest` 类相同的 `main` 方法，只不过这里 `StackComposition` 类的一个对象是继承而来的。最终，输出结果也是相同的。

---

```

1 // Class StackComposition definition with composed List object
2 class StackComposition {
3     private List s;
4
5     public StackComposition() { s = new List( "stack" ); }
6     public void push( Object o ) { s.insertAtFront( o ); }
7     public Object pop() throws EmptyListException
8     { return s.removeFromFront(); }
9     public boolean isEmpty() { return s.isEmpty(); }
10    public void print() { s.print(); }
11 }

```

---

图 17.11 使用复合类的一个简单的堆栈程序

## 17.6 队列

另一种常用的数据结构是队列。一个队列很像超级市场的收款队伍——队列中先到的人先得到服务，新加入队列的顾客只能等在最后。队列中的节点仅能从首部删除，并且仅能在尾部添加。正因为如此，我们称队列为先进先出（FIFO）的数据结构，插入和删除操作又称为入队（enqueue）和出队（dequeue）。

队列在计算机系统中有许多应用。大多数计算机只有一个简单的处理器，因此在同一时刻只能由一个用户使用。其他用户的请求就要放在一个队列中，在队列最前面的请求将是下一个服务的对象。随着队列的前进，各个用户最终得到相应的服务。

队列同样在打印缓冲中使用。在一个多用户环境中可能只有一台打印机，许多用户可能都产生需要打印的信息。如果打印机正忙，仍然可以产生其他用户的信息。这些信息将存储在磁盘的缓冲池中，在一个队列中等待，直到打印机空闲。

计算机网络中的信息数据包同样等待在队列中。每当一个数据包到达一个网络节点时，它应沿着通向这个数据包目的地的特定路径而路由到下一个节点。路由节点在同一时刻只能处理一个数据包，因此其他到达的数据包将等待在队列中直到路由器可以处理它们。

计算机网络的一个文件服务器需要处理网络中各处客户的文件存取请求。服务器处理客户请求的能力是有限的，当超出服务器的处理能力后，客户的请求将等待在队列之中。

### 常见编程错误 17.3

忘记将队列中最后一个节点的指针设置为 null

图 17.12 的程序（输出显示在图 17.13 中）继承了一个链表类以产生一个队列类。我们希望 QueueInh 类含有 enqueue 方法、dequeue 方法、isEmpty 方法和 print 方法。注意，这些方法本来是 List 类中的 insertAtBack 方法、removeFromFront 方法、isEmpty 方法和 print 方法。

```
1 // Class QueueInheritance definition
2 // Derived from List
3 class QueueInheritance extends List {
4     public QueueInheritance() { super( "queue" ); }
5     public void enqueue( Object o ) { insertAtBack( o ); }
6     public Object dequeue() throws EmptyListException
7         { return removeFromFront(); }
8     public boolean isEmpty() { return super.isEmpty(); }
9     public void print() { super.print(); }
10 }
11 // Class QueueInheritanceTest
12 public class QueueInheritanceTest {
13     public static void main( String args[] )
14     {
15         QueueInheritance objQueue = new QueueInheritance();
16
17         // Create objects to store in the queue
18         Boolean b = new Boolean( true );
19         Character c = new Character( '$' );
20         Integer i = new Integer( 34567 );
21         String s = new String( "hello" );
22
23         // Use the enqueue method
24         objQueue.enqueue( b );
25         objQueue.print();
26         objQueue.enqueue( c );
27         objQueue.print();
```

```

28         objQueue.enqueue( i );
29         objQueue.print();
30         objQueue.enqueue( s );
31         objQueue.print();
32
33         // Use the dequeue method
34         Object removedObj = null;
35
36         try {
37             while ( true ) {
38                 removedObj = objQueue.dequeue();
39                 System.out.println( removedObj.toString() +
40                                     " dequeued" );
41                 objQueue.print();
42             }
43         }
44         catch ( EmptyListException e ) {
45             System.err.println( "\n" + e.toString() );
46         }
47     }
48 }

```

图 17.12 处理一个队列

```

The queue is: true
The queue is: true $
The queue is: true $ 34567
The queue is: true $ 34567 hello
$ dequeued
The queue is: 34567 hello

34567 dequeued
The queue is: hello

hello dequeued
Empty queue

EmptyListException:The queue is empty

```

图 17.13 图 17.12 程序的输出示例

当然，链表类中还包含其他的方法（例如 `insertAtFront` 和 `removeFromBack`），它们在队列类中无法通过公有接口进行访问。记住，`List` 类中所有的公有方法在它的派生类 `QueueInheritance` 中同样也是公有方法。当实现队列中的方法时，我们让 `QueueInheritance` 类的每一个方法调用 `List` 中对应的方法——`enqueue` 调用 `insertBack`、`dequeue` 调用 `removeFromFront`、`isEmpty` 调用 `super.isEmpty` 来激活基类中的版本；`print` 也将调用 `super.print` 来激活基类中的版本。

`QueueInheritance` 类在 `QueueInheritanceTest` 的 `main` 方法中用来实例化一个称为 `objQueue` 的 `Object` 队列。其中，一个 `Boolean` 对象包含 `true`，一个 `Character` 对象包含 `$`，一个 `Integer` 对象包含 `34567`，一个 `String` 对象包含 `hello`，它们都将加入队列 `objQueue` 中，并按照先进先出的顺序出队。这些对象在一个无限循环的 `while` 中将按顺序出队。当队列中没有对象时，程序将抛出一个 `EmptyListException` 异常，并显示一条信息来说明队列为空。

## 17.7 树

链表、堆栈和队列都称为线性数据结构。树是一种非线性的、二维的、拥有特殊属性的数据结构。树的节点包含两个或更多的指针。本节讨论二叉树（如图 17.14 所示）——每个节点含有两个指针的树（全部指针或其中一个或两个都可以为 null）。根节点是树中的第一个节点，其中的每一个指针都指向一个子节点。左子节点是左子树的第一个节点，而右子节点是右子树的第一个节点。同一个节点的子节点之间称为兄弟节点。没有子节点的节点称为叶节点。计算机科学家通常自上而下地画出一棵树——正好同自然界中树木的生长方向相反。

### 常见编程错误 17.4

忘记将一棵树的叶节点的指针设置为 null

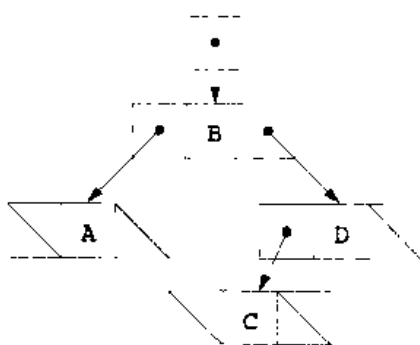


图 17.14 二叉树的图形表示

在本节中，我们将建立一种称为二叉查找树的特殊二叉树。二叉查找树（不包含同值的节点）具有这样的特性：任何左子树中的值都小于它的父节点的值，而任何右子树中的值都大于它的父节点的值。图 17.15 描绘了一个具有 12 个整数值值的二叉查找树。注意，拥有同样数据集合的一棵二叉查找树的形状是可以变化的，它依赖于这些值插入树中的顺序。

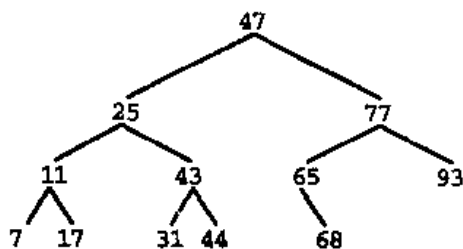


图 17.15 二叉查找树

17.16 的程序（输出示例在图 17.17 中）创建了一棵包含整数的二叉查找树，并采用三种方法来遍历此树（即走过其中所有的节点），即使用递归的中序遍历、前序遍历和后序遍历。程序产生了 10 个随机数然后将它们插入树中。

```

1 // Class TreeNode definition
2 class TreeNode {
3     TreeNode left; // left node
4     int data; // data item
5     TreeNode right; // right node
6 }

```

```

7      // Constructor: initialize data to d and make this a leaf node
8      public TreeNode( int d )
9      {
10         data = d;
11         left = right = null; // this node has no children
12     }
13
14     // Insert a TreeNode into a Tree that contains nodes.
15     // Ignore duplicate values.
16     public void insert( int d )
17     {
18         if ( d <= data )
19             if ( left == null )
20                 left = new TreeNode( d );
21             else
22                 left.insert( d );
23         else if ( d > data )
24             if ( right == null )
25                 right = new TreeNode( d );
26             else
27                 right.insert( d );
28         }
29     }
30
31 }
32
33 // Class Tree definition
34 public class Tree {
35     private TreeNode root;
36
37     // Construct an empty Tree of integers
38     public Tree() { root = null; }
39
40     // Insert a new node in the binary search tree.
41     // If the root node is null, create the root node here.
42     // Otherwise, call the insert method of class TreeNode.
43     public void insertNode( int d )
44     {
45         if ( root == null )
46             root = new TreeNode( d );
47         else
48             root.insert( d );
49     }
50
51     // Preorder Traversal
52     public void preorderTraversal() { preorderHelper( root ); }
53
54     // Recursive method to perform preorder traversal
55     private void preorderHelper( TreeNode node )
56     {
57         if ( node == null )
58             return;
59
60         System.out.print( node.data + " " );
61         preorderHelper( node.left );
62         preorderHelper( node.right );
63     }
64
65     // Inorder Traversal
66     public void inorderTraversal() { inorderHelper( root ); }
67
68     // Recursive method to perform inorder traversal

```

```

69     private void inorderHelper( TreeNode node )
70     {
71         if ( node == null )
72             return;
73
74         inorderHelper( node.left );
75         System.out.print( node.data + " " );
76         inorderHelper( node.right );
77     }
78
79     // Postorder Traversal
80     public void postorderTraversal() { postorderHelper( root ); }
81
82     // Recursive method to perform postorder traversal
83     private void postorderHelper( TreeNode node )
84     {
85         if ( node == null )
86             return;
87
88         postorderHelper( node.left );
89         postorderHelper( node.right );
90         System.out.print( node.data + " " );
91     }
92 }
93 // This program tests the Tree class.
94 import java.util.*;
95
96 // Class TreeTest definition
97 public class TreeTest {
98     public static void main( String args[] )
99     {
100         Tree tree = new Tree();
101         int intVal;
102
103         System.out.println( "Inserting the following values: " );
104
105         for ( int i = 1; i <= 10; i++ ) {
106             intVal = (int) ( Math.random() * 100 );
107             System.out.print( intVal + " " );
108             tree.insertNode( intVal );
109         }
110
111         System.out.println ( " \n \n Preorder traversal" );
112         tree.preorderTraversal();
113
114         System.out.println ( " \n \n Inorder traversal" );
115         tree.inorderTraversal();
116
117         System.out.println ( " \n \n Postorder traversal" );
118         tree.postorderTraversal();
119         System.out.println();
120     }
121 }

```

图 17.16 创建并遍历二叉查找树

让我们分析图 17.16 中的二叉树程序。TreeTest 类的 main 方法在开始时将实例化一个 Tree 类的 tree 对象（第 100 行）。程序随机产生 10 个整数，每个整数通过调用 insertNode 方法（第 108 行）而插入到二叉查找树中。然后程序对这棵树进行前序、中序和后序遍历（后面将简要介绍）。

```

Inserting the following values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39

```

图 17.17 图 17.16 的程序输出示例

现在，我们来详细分析类的定义和各种方法。我们从 `TreeNode` 类（第 1 行）开始，它定义的友元数据包括节点的 `data` 值、引用 `left`（指向节点的左子树）、引用 `right`（指向右子树）。构造函数（第 8 行）将其参数传递给 `data` 作为它的值，同时将引用 `left` 和 `right` 设置为 `null`（这样我们就将查找树初始化为一个叶节点）。`Tree` 类的 `insertNode` 将激活 `insert` 方法，从而将数据加入到一个非空的树中。下面将详细讨论这个方法。

`Tree` 类拥有一个私有的数据成员 `root`，一指向树的根节点的一个引用。这个类拥有公有的 `insertNode` 方法（在树中插入一个节点），以及 `preorderTraversal` 方法、`inorderTraversal` 方法和 `postorderTraversal` 方法，后面的三个方法各采用一种不同的方式对树进行遍历。这三个方法都调用它们自己不同的递归方法来对二叉树进行相应的内部操作。`Tree` 的构造函数将 `root` 初始化为 `null`，这表明最初的树为空。

`Tree` 类的 `insertNode` 方法（第 43 行）首先判断树是否为空。如果是，则该方法将分配一个新的 `TreeNode`，使用将要插入树中的整数对其进行初始化，然后将这个新的节点赋给引用 `root`。如果树不为空，则将递归地调用 `TreeNode` 的 `insert` 方法，以便在树中加入新的节点。在一棵二叉查找树中，新的节点仅能作为叶节点插入。

`TreeNode` 的 `insert` 方法将想要插入的值同根节点中的 `data` 值进行比较。如果插入的值小于或等于根节点的值，程序将判断根的左子树是否为空（第 19 行）。如果是，则将分配一个新的 `TreeNode`，并使用将要插入的数据对其进行初始化，同时把引用 `left` 指向新的节点（第 20 行）；否则，`insert` 递归调用自己（第 22 行），目的是在左子树中搜寻到可以插入的节点。如果将要插入的值大于根节点的值，程序就判断右子树是否为空（第 25 行）。如果是，则将分配一个新的 `TreeNode`，并使用将要插入的数据对其进行初始化，同时把引用 `right` 指向新的节点（第 26 行）；否则，`insert` 递归调用自己（第 28 行），目的是在右子树中查找到可以插入的节点。

`inorderTraversal`、`preorderTraversal` 和 `postorderTraversal` 方法调用帮助方法 `inorderHelper`、`preorderHelper` 和 `postorderHelper`，分别遍历树（如图 17.18 所示）并打印节点的值。

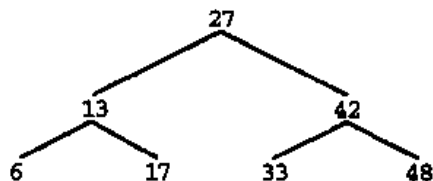


图 17.18 二叉查找树

`inorderTraversal`（第 66 行）方法的执行步骤是：

1. 调用 `inorderHelper` 方法（第 69 行）来遍历左子树
2. 处理节点中的内容（即打印节点的值）
3. 调用 `inorderHelper` 来遍历右子树。

直到遍历完一个节点的左子树后再处理该节点的内容。按照 `inorderTraversal` 方法遍历图 17.18 的查找树的结果为：

6    13    17    27    33    42    48

注意，一棵二叉查找树按 `inorderTraversal` 方法遍历的结果是升序的。创建一棵二叉查找树的过程实际上对数据进行了排序处理。这种处理过程称为二叉树排序。

`preorderTraversal` 方法（第 52 行）的执行步骤是：

1. 处理节点中的内容。
2. 调用 `preorderHelper` 方法（第 55 行）来遍历左子树。
3. 调用 `preorderHelper` 来遍历右子树。

每当遍历到一个节点时就首先访问该节点的内容，处理完指定节点的值后，将首先遍历左子树的值，然后遍历右子树的值。按照 `preorderTraversal` 方法遍历图 17.18 的查找树的结果是：

27    13    6    17    42    33    48

`postorderTraversal` 方法（第 80 行）的执行步骤是：

1. 调用 `postorderHelper`（第 83 行）来遍历左子树
2. 调用 `postorderHelper` 来遍历右子树。
3. 处理节点中的内容。

只有处理完一个节点的所有子树之后，才将处理这个节点的内容。按照 `postorderTraversal` 方法遍历图 17.18 的查找树的结果是：

6    17    13    33    48    42    27

二叉查找树可以去除同值的节点。当我们创建一棵树时，插入与树中某一元素相同的值是可以判断出来的，这个值和其他的值一样可以遵循“向左”或者“向右”这样的判断结果。因此，这个值最终将同树中原有的同值元素进行一次比较。只需将重复的值简单地删除即可。

在一棵二叉树中搜寻某个数值是相当快的，特别是在满二叉树中。在一棵满二叉树中，每一级节点的数目都是上一级的两倍，图 17.18 就是一棵满二叉树。因此一棵拥有  $n$  个元素的二叉查找树最少有  $\log_2 n$  层，这相当于仅需要  $\log_2 n$  次比较，就可以在二叉树中确定存在或不存在某个数值。例如，当我们在一个具有 1 000 个元素的满二叉查找树中搜索某一数值时，大约进行 10 次比较就可以获得结果，因为  $2^{10} > 1\,000$ 。当我们在一个具有 1 000 000 个元素的满二叉查找树中搜索某一数值时，大约进行 20 次比较就可以满足要求，因为  $2^{20} > 1\,000\,000$ 。

在本章的练习中，我们将会看到其他几种二叉树的操作，例如从二叉树中删除一个节点，采用一种二维树的格式打印一棵二叉树，以及在一棵二叉树上进行按层遍历。所谓按层遍历二叉树是指从一棵树的根节点开始，一行一行地遍历它的节点。其他有关二叉树的练习还包括建立一棵允许具有同值节点的二叉查找树，在一棵二叉树中插入字符串的值，以及判断一棵二叉树含有多少层。



## 小结

- 自引用的类包含称为指针的成员，它们指向相同类型的对象。
- 自引用的类能够使很多自引用的对象链接在一起，从而构成堆栈、队列、链表和树。
- 动态内存请求在内存中预留一块区域，以便在程序运行期间存放一个对象。
- 链表是一些自引用对象的线性集合。
- 链表是动态数据结构——链表的长度可以根据需要增加或减少。
- 链表可以一直扩充，直到内存耗尽为止。
- 通过引用处理，链表可以提供一种快速插入和删除数据的机制。
- 堆栈和队列是链表的压缩版本。
- 只能在堆栈的顶部进行节点的添加和删除操作。正因为如此，我们称堆栈是一种后进先出（LIFO）的数据结构。
- 堆栈的最后一个节点的链接成员将设置成 null，用来表明堆栈的底部。
- 操作堆栈的两种主要方法是 push 和 pop。push 操作创建一个新节点并把它放在堆栈的顶部，pop 操作将堆栈顶部的节点删除并返回弹出的值。
- 队列这种数据结构从头部删除节点，在尾部添加节点。正因为如此，我们称队列是一种先进先出（FIFO）的数据结构，添加和删除操作由 enqueue 和 dequeue 方法完成。
- 树是二维数据结构，其中每个节点都需要有两个或更多的指针。
- 二叉树的每个节点有两个指针。
- 树中的第一个节点称为根节点。
- 根节点中的每一个指针都指向一个子节点。左子节点是左子树的第一个节点，而右子节点是右子树的第一个节点。同一个节点的子节点之间称为兄弟节点，没有子节点的节点称为叶节点。
- 一棵二叉查找树具有这样的特性：一个节点的左子节点的值小于它的父节点的值，而一个节点的右子节点的值大于或等于它的父节点的值。如果数据值都互不相等，那么右子节点的值就只是大于它的父节点的值。
- 二叉树的中序遍历首先中序遍历左子树，处理根节点的内容，然后中序遍历右子树，一般直到一个节点的左子树的所有值处理完成后再处理该节点的内容。
- 二叉树的前序遍历首先处理根节点的内容，前序遍历左子树，然后前序遍历右子树，一般每当遍历到一个节点时就首先处理该节点的内容。
- 二叉树的后序遍历首先后序遍历左子树，后序遍历右子树，然后处理根节点的内容，一般直到一个节点的两棵子树的所有值处理完成后再处理该节点的内容。

## 术语

binary search tree 二叉查找树

binary tree 二叉树

binary tree sort 二叉树排序

child node 子节点

children 子节点

delete a node 删除一个节点

dequeue

duplicate elimination 删除重复节点

dynamic data structures 动态数据结构

enqueue

FIFO(first-in, first-out) FIFO (先进先出)	postorder traversal of a binary tree 后序遍历二叉树
head of a queue 队列的头部	preorder traversal of a binary tree 前序遍历二叉树
inorder traversal of a binary tree 二叉树的中序遍历	push
insert a node 插入一个节点	queue 队列
leaf node 左节点	recursive tree traversal algorithms 递归遍历树的算法
left child 左子节点	right child 右子节点
left subtree 左子树	right subtree 右子树
level-order traversal of a binary tree 二叉树的按层遍历	root node 根节点
LIFO(last-in, first-out) LIFO (后进先出)	self-referential class 自引用的类
linear data structure 线性数据结构	stack 堆栈
linked list 链表	subtree 子树
node 节点	tail of a queue 队列的尾部
nonlinear data structure 非线性数据结构	top of a stack 堆栈的顶部
null reference null 引用	traversal 遍历
parent node 父节点	tree 树
pop	visit a node 访问一个节点

## 自测练习

### 17.1 填空:

- 一个自我\_\_\_\_\_的类用来建立动态数据结构, 这样的结构可以在执行时动态增长或缩小。
- \_\_\_\_\_运算符用来动态分配内存空间, 这个运算符返回指向分配了内存的引用。
- \_\_\_\_\_是一个压缩版本的链表, 其中节点的删除和插入仅能在表头进行; 这种数据结构按后进先出的顺序返回节点。
- 不修改链表而仅仅简单地查看链表是否为空的方法, 称为\_\_\_\_\_方法。
- 我们将队列称为\_\_\_\_\_的数据结构, 因为首先插入的节点将首先删除。
- 指向链表中下一个节点的引用称为\_\_\_\_\_。
- 在 Java 中, 自动回收动态分配的内存称为\_\_\_\_\_。
- \_\_\_\_\_是一个压缩版本的链表, 其中节点仅能在链表的底部插入, 从链表的头部删除。
- \_\_\_\_\_是非线性、二维的数据结构, 它的节点包含两个或更多的指针。
- 堆栈被认为是一种\_\_\_\_\_的数据结构, 因为最后插入的节点位置正好是最先删除的节点位置。
- 一棵树的\_\_\_\_\_节点包含两个链接成员。
- 一棵树的第一个节点称为\_\_\_\_\_节点。
- 一棵树中节点的每个指针分别指向这个节点的\_\_\_\_\_和\_\_\_\_\_。
- 指针为空的树中的节点称为\_\_\_\_\_节点。
- 本章介绍的对于二叉树进行遍历的 4 种算法是\_\_\_\_\_。

- 17.2 链表和堆栈的区别是什么?
- 17.3 堆栈和队列的区别是什么?
- 17.4 也许本章更确切的一个标题应该是“可重用的数据结构”。说明下面这几个机制在实现数据结构可重用性方面的作用:
- 类
  - 继承
  - 复合
- 17.5 手工给出图 17.19 中二叉查找树进行中序、前序和后序遍历的结果。

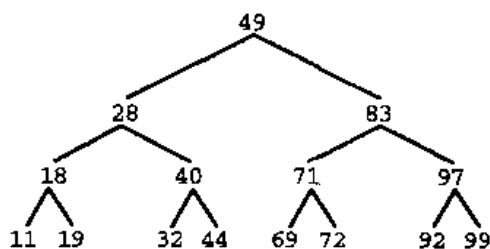


图 17.19 拥有 15 个节点的二叉查找树

## 自测练习答案

- 17.1 a) 引用。b) new。c) 堆栈。d) 谓词。e) 先进先出。f) 指针。g) 无用单元回收。h) 队列。i) 树。j) 后进先出。k) 二叉。l) 根。m) 子节点或子树。n) 叶。o) 中序、前序、后序和按层遍历。
- 17.2 可以在一个链表的任何位置插入或删除一个节点。注意，一个堆栈仅允许在堆栈的顶部插入、从堆栈的顶部删除。
- 17.3 一个队列既拥有指向头部的引用，也拥有指向尾部的引用。这样我们可以在尾部增加节点并且从头部删除节点。一个堆栈仅需要一个指向顶部的引用，因为插入和删除操作都在顶部进行。
- 17.4 a) 类允许我们按照自己的需要来实例化某一种类型（即类）尽可能多的对象。
- b) 继承能使我们在一个子类中重用父类的代码，这使得派生类的数据结构同时也是基类的数据结构。
- c) 复合能使我们在一个复合类中重用作为其数据成员的另一个类的代码；如果我们将这个作为数据成员的类对象标识为私有，那么通过复合类的接口是无法访问到这个对象中的公有方法。
- 17.5 中序遍历的结果是：

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

前序遍历的结果是：

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

后序遍历的结果是：

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

## 练习

- 17.6 编写一个程序，链接两个字符链表的对象。ListConcat 类中应该包含 concatenate 方法，该方法将两个链表对象的引用作为参数，并将第二个链表链接到第一个链表上。
- 17.7 编写一个程序，将两个有序的整数链表合并为一个有序的整数链表。ListMerge 类的 merge 方法将合并前两个链表的引用作为参数，返回合并后链表的引用。
- 17.8 编写一个程序，按照顺序将 25 个 0 到 100 之间随机产生的整数插入到一个链表中。程序应该可以计算这些元素的和，以及它们平均值的浮点结果。
- 17.9 编写一个程序，创建一个拥有 10 个字符的链表，然后创建第二个链表使其包含第一个链表的副本，但顺序是相反的。
- 17.10 编写一个程序，接收一行文本输入，然后使用一个堆栈来将它们反向输出。
- 17.11 编写一个程序，使用一个堆栈来判断一个字符串是否为回文（即字符串从前向后和从后向前拼写都是相同的）。程序可以忽略空格和标点符号。
- 17.12 堆栈在编译器中用来帮助分析表达式和产生机器码。在本练习和下一个练习中，我们研究编译器如何分析仅具有常量、运算符和括号的表达式。

人们常常将表达式写成：3+4 或 7/9。其中运算符（本例中为 + 和 /）写在操作数值之中——我们称为中缀表示法。计算机更“喜欢”后缀表示法，其中运算符写在其两个操作数的右方。上述的中缀表达式写成后缀表达式应分别为 34+ 和 79/。

为了分析一个复杂的中序表达式，编译器首先应该将表达式转化为后序表示，然后分析这个表达式的后序版本。这些算法仅需对表达式进行一次从左向右的扫描。每一个算法使用一个堆栈来支持它的操作，并且在每个算法中堆栈的用途是不同的。

在本练习中，要求编写一个 Java 版本的中序到后序的表达式转化算法。在下一个练习中，要求写出一个 Java 版本的后序表达式求值算法。在本章的最后，可以使用在本练习中所编写的代码，帮助实现一个完整的、可工作的编译器。

编写 InfixToPostfixConverter 类，它可以仅包含单一整数字符的传统中序算术表达式（假设这样的表达式都是合法的）转化为一个后序表达式，如下所示：

(6 + 2) \* 5 - 8 / 4

上述表达式的后序版本为：

6 2 + 5 \* 8 4 / -

程序应该将表达式读入 StringBuffer infix 中，然后使用本章介绍过的某个堆栈类来帮助创建存放在 StringBuffer postfix 中的后序表达式。下面给出创建后序表达式的算法。

- 1) 将一个左括号 '(' 压入栈中。
- 2) 将一个右括号 ')' 放置在 infix 的尾部。
- 3) 如果堆栈不为空，从左向右读 infix，并进行下列步骤：

如果 infix 中的当前字符是一个数字，将它复制给 postfix 的尾部。

如果 infix 中的当前字符是一个左括号，将它压入堆栈。

如果 infix 中的当前字符是一个运算符：

若堆栈顶部的运算符（如果存在）优先级等于或高于当前运算符，则弹出堆栈顶部

的运算符, 然后将其接在 postfix 的尾部; 将 infix 中的当前运算符压入堆栈。

如果 infix 中的当前字符是一个右括号:

将运算符从堆栈顶部弹出, 然后将它们接在 postfix 尾部, 直到堆栈顶部出现一个左括号; 将左括号从堆栈顶部弹出并删除。

表达式中允许使用下列算术运算符:

+ 加  
- 减  
\* 乘  
/ 除  
^ 幂  
% 取模

堆栈的每个节点应该包含一个实例变量以及指向下一个节点的引用。这里可能需要使用如下的一些方法:

- convertToPostfix 方法用来将中序表达式转化为后序表达式。
- isOperator 方法用来判断参数 c 是否为一个运算符。
- precedence 方法用来判断运算符 operator1 (在中序表达式) 的优先级是大于、等于还是小于 operator2 (在堆栈中) 的优先级。如果 operator1 的优先级小于 operator2, 则返回 true; 否则返回 false。
- stackTop 方法 (应该将其加入堆栈类中) 在不将节点弹出堆栈的前提下返回堆栈顶端的值。

17.13 编写一个 PostfixEvaluator 类, 用来计算一个下述类型的后缀表达式 (假设表达式是合法的):

6 2 + 5 \* 8 4 / -

程序应该可以将一个包含数字和运算符的后缀表达式读入一个 StringBuffer。使用本章讨论过的堆栈方法的修改版本, 程序应该能够扫描并计算表达式。下面给出这个算法。

- 1) 将一个右括号 ')' 放置在后缀表达式的尾部。如果遇到这个右括号字符, 那么处理终止。
- 2) 如果没有读到右括号, 就从左向右读取表达式:  
如果当前字符为数字, 将它的整数值压入栈中 (一个数字字符的整数值相当于它在计算机字符集中的编码值减去字符 '0' 在字符集中的编码值)。  
否则, 如果当前字符为一个运算符:  
将堆栈中最顶端的两个值弹出, 分别赋给变量 x 和 y;  
计算  $y \text{ operator } x$ ;  
将计算结果压入栈中。
- 3) 如果遇到表达式中的右括号, 就将堆栈顶端的值弹出。这就是后缀表达式的计算结果。

注意, 在 2) 中, 如果运算符为 '/', 堆栈的顶端为 2, 并且堆栈中下一个元素为 8, 那么首先将 2 弹出赋给 x, 将 8 弹出赋给 y, 计算  $8/2$ , 结果为 4 将压入栈中。对于运算符 '-' 同样使用。在合法表达式中允许出现的算术运算符为:

+ 加  
- 减

\* 乘  
/ 除  
^ 幂  
% 取模

这个堆栈应该使用本章介绍的一种堆栈类来实现,本练习应实现下述方法:

- a) 使用 `evaluatePostfixExpression` 方法计算后缀表达式的值
- b) 使用 `calculate` 方法计算表达式 `op1 operator op2`
- c) 使用 `push` 方法将一个值压入堆栈
- d) 使用 `pop` 方法将一个值从堆栈中弹出
- e) 使用 `isEmpty` 方法判断堆栈是否为空
- f) 使用 `printStack` 方法打印堆栈。

17.14 修改习 17.13 中的后缀表达式计算程序,使它能够处理大于 9 的整数。

17.15 (超级市场模拟) 编写一个程序,模拟一个超级市场的收款队列。收款队列用一个队列对象实现。顾客(即客户对象)到达队列的时间间隔是 1 到 4 分钟之间的一个随机整数。同时,每个顾客接受服务的时间也是 1 到 4 分钟之间的一个随机整数。显然,我们需要平衡这个速率。如果到达的平均速率大于接受服务的平均速率,队列的长度就会无限增长。有时,即使采用“平衡”速率,由于随机的关系同样会导致很长的队列。采用下列算法编写超级市场程序,连续运行 12 个小时(720 分钟)。

- 1) 选择一个 1 到 4 之间的随机整数,确定第一个顾客到达的时间。
- 2) 当第一个顾客到达时:
  - 判断顾客的服务时间(随机产生从 1 到 4 的一个整数);
  - 开始对顾客进行服务;
  - 预计下一个顾客到达的时间(一个 1 到 4 的随机整数同当前的时间相加)。
- 3) 在这一天中的每一分钟:
  - 如果下一个顾客到达,将顾客增加到队列中,预计下一个顾客的到达时间;
  - 如果最近一个顾客的服务已经结束,让队列中的下一个顾客出队,确定对这个顾客进行服务的完成时间(一个 1 到 4 的随机整数同当前时间相加)。

现在连续运行模拟程序 720 分钟,并回答下面一系列问题:

- a) 任意时刻,队列中顾客的最大数目是多少?
  - b) 对于任意顾客而言,等待的最长时间是多少?
  - c) 如果顾客到达的时间间隔从 1 到 4 分钟改为 1 到 3 分钟,那么将发生什么情况?
- 17.16 修改图 17.16 所示的程序,使得二叉树中的节点可以包含相同的值。
- 17.17 编写一个基于图 17.16 的程序,接收一行文本的输入,然后将每个句子分隔成单独的单词(这里可能需要用到 `java.io` 软件包中的 `StreamTokenizer` 类),将这些单词插入一个二叉查找树中,然后按中序、前序和后序打印这棵树。
- 17.18 在本章中,可以看到在创建一棵二叉查找树时,需要简单地删除同值的节点。描述如何仅使用一个单下标数组来删除同值的节点,比较使用数组和使用二叉查找树进行同值节点的删除时的不同。
- 17.19 编写一个 `depth` 方法,接收一棵二叉树作为参数并判断它有多少层。
- 17.20 (递归地从后向前打印一个表) 编写一个 `PrintListBackwards` 方法,递归地按照逆序输出一个

链表中的各项。编写一个测试程序,能够创建一个排过序的整数表,并能逆序打印该表。

- 17.21 (递归地从后向前搜索一个表) 编写一个 `searchList` 方法,用来递归地从后向前搜索一个链表中某一特定的值。如果找到这个值,就返回相应节点的引用;否则返回空指针。在一个建立整数链表的程序中使用这一方法。程序将提示用户输入一个值并在表中对其进行定位。

- 17.22 (二叉树的删除) 在本练习中,我们讨论如何从二叉查找树中删除一个元素。删除算法同插入算法相比没有那样直接。当删除一个元素时将遇到三种情况——这个元素包含在一个叶节点中(即这个节点没有子节点),这个元素所在的节点仅拥有一个子节点,或者这个节点拥有两个子节点。

如果被删除元素包含在叶节点中,就简单地将这个节点删除,并将其父节点中相应的引用设置为空。

如果这个元素所在节点拥有一个子节点,那么这一被删除节点的父节点的引用就设置为指向这一子节点,并将元素所在的节点删除。这也意味着子节点将取代被删除节点在树中的位置。

最后一种情况最为复杂,当被删除节点拥有两个子节点时,树中另外的一个节点将取代这个节点的位置。尽管如此,父节点的指针不能简单地指向被删除节点的两个子节点之一。在大多数情况下,这样生成的二叉树就无法保持二叉查找树的特性(不考虑包含同值节点的情况):任何左子树中节点的值小于父节点的值,同时任何右子树中节点的值大于父节点的值。

那么选择哪一个节点取代被删除节点且又能保持二叉树的特性呢?或者这个节点包含树中小于被删除节点中元素值的最大元素值,或者这个节点包含树中大于被删除节点中元素值的最小元素值。让我们考虑包含较小元素值的情况。在一棵二叉查找树中,小于一个父节点元素值的最大值位于这个父节点的左子树中,并且它就是这棵左子树中最右边的那个节点。

可以通过向右遍历这棵左子树,直到当前节点的右节点为空来得到这个节点。我们现在考虑这个用来替换的节点是一个叶节点或拥有左子节点的情况。如果这个替换节点就是一个叶节点,则删除的操作如下:

- 1) 将被删除节点的引用赋给一个临时引用变量。
- 2) 将被删除节点的父节点指向左子树的引用指向替换节点。
- 3) 将替换节点的父节点中相应的引用置为空。
- 4) 将被删除节点的右子树赋给替换节点的右子树。
- 5) 将被删除节点的左子树赋给替换节点的左子树。
- 6) 删除临时引用变量指向的节点。

对于拥有一个左子节点的替换节点,删除步骤同上述基本相同。但是在算法中必须将这个子节点移到替换节点原有的位置上。如果替换节点拥有一个左子节点,则删除操作如下所示:

- 1) 将被删除节点的引用赋给一个临时引用变量。
- 2) 将被删除节点的父节点指向左子树的引用指向替换节点。
- 3) 将替换节点的父节点中指向右子树的引用指向替换节点的左子节点。

- 4) 将被删除节点的右子树赋给替换节点的右子树。
- 5) 将被删除节点的右子树赋给替换节点的右子树。
- 6) 删除临时引用变量指向的节点。

编写一个 `deleteNode` 方法，把要删除的元素作为参数传递进来。该方法应该首先在树中找到包含这个元素的节点，然后使用这里讨论的算法删除这一节点。如果在树中没有发现包含这一元素的节点，则给出一条信息表明这个值无法删除。修改图 17.16 中的程序来实现这一方法。在删除操作完成之后，调用 `inorderTraversal` 方法、`preorderTraversal` 方法和 `postorderTraversal` 方法来确认节点删除是否正确。

- 17.23 ( 二叉树搜索 ) 编写一个 `binaryTreeSearch` 方法，在一棵二叉查找树中定位某个特定的节点值，搜寻的数值将作为方法的参数。如果搜寻到包含这个数值的节点，应该返回这个节点的引用；否则返回一个空指针。
- 17.24 ( 按层遍历二叉树 ) 图 17.16 的程序描述了三种遍历二叉树的递归方法——中序、前序和后序遍历。本练习将按层遍历二叉树，即从根节点开始一层一层打印出节点的值，每一层的节点从左向右遍历。按层遍历的算法并非一个递归的算法，它使用了一个队列对象来控制节点的输出，下面给出了这个算法。

- 1) 将根节点插入队列。
- 2) 如果队列中还有节点：
  - 则取得队列中下一个节点，打印节点的值；
  - 如果这个节点指向左子树的引用不为空，将左子树插入队列；
  - 如果这个节点指向右子树的引用不为空，将右子树插入队列。

编写一个 `levelOrder` 方法，按层遍历一棵二叉树的对象。修改图 17.16 中的程序以便使用本方法（注意：可能还需要用到图 17.12 中的队列处理方法）。

- 17.25 ( 打印树 ) 编写一个递归的 `outputTree` 方法，在屏幕上显示一棵二叉树，该方法应该自顶向下逐行地在屏幕上从左向右显示这棵树。每一行都纵向排列。例如，图 17.19 中的二叉树按照下图的格式输出：

```

          99
        97  92
      83    72
    71     69
  49      44
    40    32
    28    19
    18    11

```

注意，树中最右方的叶节点显示在屏幕最右一列的最上方，而根节点显示在最左方。`outputTree` 方法应该接收一个 `totalSpaces` 作为参数，它代表输出时放在数值前的空格数目（这个变量的值应该从零开始，这样根节点可以出现在屏幕的左方）。该方法使



用了一种改动的中序遍历方法来输出这棵树——它从树中最右方的节点开始然后向左前进。算法如下：

当指向当前节点的引用不为空时，

    使用当前节点的右子树以及 `totalSpace+5` 作为参数递归调用 `outputTree`；

    使用一个从 1 到 `totalSpace` 的 `for` 结构来输出空格；

    输出当前节点的值；

    将指针指向当前节点的左子树；

    将 `totalSpace` 的值增加 5。

## 特殊小节：建立自己的编译器

在练习 5.42 和练习 5.43 中，我们介绍了 Simpletron 机器语言 (SML)，并且让读者实现了一个可以执行 SML 程序的 Simpletron 计算机模拟器。在本节中，我们将建立一个能够将高级语言编写的程序转化为 SML 的编译器。这一节的内容与整个程序处理过程联系紧密，我们要使用一种新的高级语言编写程序，并使用自己建立的编译器来编译这些程序，然后在练习 5.43 建立的模拟程序中运行这些程序。读者应尽量使用面向对象的技术来完成自己的编译器。

17.26 (Simple 语言) 在开始建立自己的编译器之前，我们首先讨论一种简单的、但是功能强大的高级语言，它有些类似通用 Basic 语言的早期版本，我们把这种语言称为 Simple。每一条 Simple 的语句包括行号和一条 Simple 指令，行号应该按升序排列。每一个指令都是从下述的 Simple 命令之一开始的：`rem`、`input`、`let`、`print`、`goto`、`if/goto` 或 `end` (如图 17.20 所示)。除了 `end` 之外，所有的命令都可以多次重复使用。Simple 仅仅计算使用 `+`、`-`、`*` 和 `/` 的整数表达式。这些运算符使用和 Java 相同的优先级顺序。可以在表达式中使用括号来改变计算顺序。

命令	例句	描述
<code>rem</code>	50 <code>rem this is a remark</code>	跟在 <code>rem</code> 后的任何语句都将作为注释，并且由编译器忽略
<code>input</code>	30 <code>input x</code>	显示一个问号来提示用户输入一个整数。从键盘读入这个整数并将其存储在 <code>x</code> 中
<code>let</code>	80 <code>let u = 4 * (j-56)</code>	将 <code>4*(j-56)</code> 的值赋给 <code>u</code> 。注意，任何复杂的表达式都可能出现在等号的右方
<code>print</code>	10 <code>print w</code>	显示 <code>w</code> 的值
<code>goto</code>	70 <code>goto 45</code>	将程序控制转到第 45 行
<code>if/goto</code>	35 <code>if i == z goto 80</code>	比较 <code>i</code> 和 <code>z</code> 是否相等，如果相等则将程序控制转到第 80 行；否则继续执行下一条指令
<code>end</code>	99 <code>end</code>	结束程序执行

图 17.20 Simple 命令

这个简单的编译程序仅仅能够分辨小写字母，所有在 Simple 文件中出现的字符应该都是小写的 (除非大写字母出现在 `rem` 语句中，程序将它们视为注释而忽略，否则将导致一个语法错误)。一个变量名是一个单独的字母。Simple 不允许声明变量，因此应该在注释中指出程序中使用到的变量。Simple 只使用整数变量，并且没有变量声明部分。只要在程序中给出一个新的变量名，那么就认为已对这个变量进行了声明并自动将其初始化为零。Simple 的语法不允许进行字符串操作 (字符串的读、写和比较字符串等)。如果在 Simple 程序中出现字符串 (除非出现在 `rem` 之后)，编译器将产生语法错误。练习 17.29 要求学生修改编译器以便它可以进行语法错误检查。

Simple 使用条件 `if/goto` 语句和无条件的 `goto` 语句在程序运行过程中改变控制流。如果 `if/goto` 语句中的条件为 `true`，则将控制转移到程序中的某一特定行。下述关系运算符

及相等运算符在一个 if/goto 语句中是合法的: <、>、<=、>=、== 或 !=, 这些运算符的优先级同 Java 中一致。

现在让我们考虑几个演示 Simple 特性的程序。第一个程序 (如图 17.21 所示) 从键盘读取两个整数, 并将值存储在变量 a 和 b 中, 然后计算并打印它们的和 (存储在变量 c 中)。图 17.22 中的程序判断并打印两个整数中较大的一个, 这些整数将从键盘输入并存储在 s 和 t 中。if/goto 语句测试条件  $s \geq t$ , 如果条件为 true, 则将控制转移到第 90 行并输出 s 的值; 否则输出 t, 并将控制转移到第 99 行的 end 语句, 从而终止程序。

---

```

1  10 rem determine and print the sum of two integers
2  15 rem
3  20 rem input the two integers
4  30 input a
5  40 input b
6  45 rem
7  50 rem add integers and store result in c
8  60 let c = a+b
9  65 rem
10 70 rem print the result
11 80 print c
12 90 rem terminate program execution
13 99 end

```

---

图 17.21 计算两个整数和的 Simple 程序

---

```

1  10 rem determine and print the larger of two integers
2  20 input s
3  30 input t
4  32 rem
5  35 rem test if s >= t
6  40 if s >= t goto 90
7  45 rem
8  50 rem t is greater than s, so print t
9  60 print t
10 70 goto 99
11 75 rem
12 80 rem s is greater than or equal to t, so print s
13 90 print s
14 99 end

```

---

图 17.22 寻找两个整数中较大一个的 Simple 程序

Simple 不提供循环结构 (例如 Java 中的 for、while 或 do/while), 尽管如此, Simple 仍能利用 if/goto 和 goto 语句来实现 Java 中的任一种循环结构。图 17.23 使用了一种标记控制循环来计算几个整数的平方, 从键盘输入整数并存储在变量 j 中。如果输入的值等于标记 -9999, 则将控制转移到程序结束的第 99 行; 否则将 j 的平方赋给 k 并输出到屏幕上, 并且控制跳到第 20 行等待输入下一个整数。使用图 17.21、图 17.22、图 17.23 中的例程作为编程指导, 编写一个 Simple 程序, 实现下述每一种功能:

- a) 输入三个整数, 计算它们的平均值并打印结果。
- b) 采用标记控制循环输入 10 个整数, 计算并打印它们的和。
- c) 采用计数器控制循环输入 7 个整数, 包括正数和负数, 计算并打印它们的平均值。

- d) 输入一系列整数, 判断并打印其中最大的一个。第一个输入的整数表示一共对多少个  
数进行操作。
- e) 输入 10 个整数并打印其中最小的一个。
- f) 计算并打印 2 到 30 之间所有偶数的和。
- g) 计算并打印 1 到 9 之间所有奇数的积。

```

1  10 rem calculate the squares of several integers
2  20 input j
3  23 rem
4  25 rem test for sentinel value
5  30 if j -- -9999 goto 99
6  33 rem
7  35 rem calculate square of j and assign result to k
8  40 let k = j * j
9  50 print k
10 53 rem
11 55 rem loop to get next j
12 60 goto 20
13 99 end

```

图 17.23 计算几个整数的平方

#### 17.27 (建立一个编译器。要求先完成练习 5.42、练习 5.43、练习 17.12、练习 17.13 和练习 17.26)

现在, 我们已经有了 Simple 语言 (练习 17.26), 下面来讨论如何建立一个 Simple 编译器。首先, 我们考虑一个 Simple 程序转换成 SML 并由 Simpletron 模拟程序执行的过程 (如图 17.24 所示)。编译器读取一个包含 Simple 程序的文件, 并将其转换成 SML 代码, SML 代码将输出到磁盘上的一个文件, 其中每一行都有 SML 指令。然后将 SML 文件载入 Simpletron 模拟程序, 将其结果送到磁盘上的一个文件并输出到屏幕。注意, 练习 5.19 开发的 Simpletron 程序是从键盘得到输入的, 我们必须将其修改为通过读取一个文件来得到输入, 以便它能够运行由自己编写的编译器产生的程序。

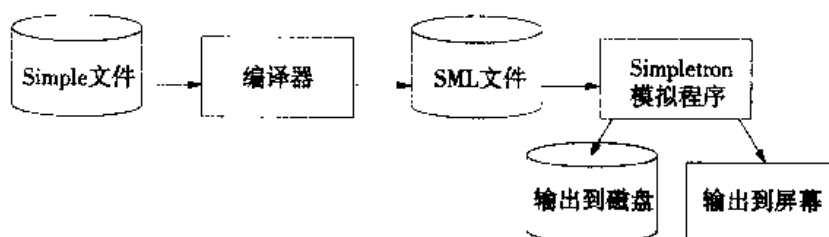


图 17.24 编写、编译和执行 Simple 程序

图 17.24 中的 Simple 编译器通过对 Simple 程序的两遍扫描来将它转换成 SML。第一步编译创建一个符号表 (对象), 其中存放有该 Simple 程序的每个行号 (对象)、变量名 (对象) 和常量 (对象), 以及它们的类型和在最后的 SML 代码中的存储单元 (在下面的叙述中将讨论符号表)。第一步编译也产生每条 Simple 语句 (对象) 对应的 SML 指令对象。如果 Simple 程序包含将控制转移到其后行号的语句, 那么第一步编译将导致 SML 程序中包含一些“未完成”的指令。编译器的第二遍扫描定位并完成这些“未完成”的指令, 将 SML 程序输出到一个文件中。

## 第一步编译

一开始,编译器将Simple程序的一条语句读入内存。为了操作和编译,这一行必须分解为单个的标记(即一条语句的片段,可以利用java.io软件包中的StreamTokenizer类来实现)。然后检索每条以行号开始并且后面跟着一个命令的语句。当编译器将一条语句分解为标记时,如果标记是一个行号、一个变量或者一个常量,就将其放到符号表中。仅当行号是一条语句的第一个标记时,编译器才把它放到符号表中。symbolTable对象是一个tableEntry对象的数组,而每个tableEntry对象都是程序中的符号。由于不限制程序中出现的符号的数量,因此一个特定程序的symbolTable就会很大。现在我们建立一个symbolTable,它是一个100个元素的数组。在程序运行时,可以增大或减小它的规模。

每个tableEntry对象包括三个成员。成员symbolI是表示一个变量、行号或常量的ASCII码的整数(这些变量名都是单个字符),成员type是指示符号类型的下述字符之一:'C'表示常量,'I'表示行号,'V'表示变量。成员location是符号所指向的Simpletron的存储单元(00到99)。Simpletron内存是100个整数的一个数组,其中存放着SML指令和数据。对于一个行号来说,它的存储单元是Simpletron内存数组的一个元素,表示与Simple语句对应的SML指令开始的存储单元。对变量或常量来说,其存储单元是存储该常量或变量的Simpletron内存数组的一个元素。变量和常量从Simpletron内存的末尾向前分配。第一个变量或常量存放在位置99,下一个存放在98,依次类推。

符号表在Simple程序转换到SML的过程中是作为一个整体来处理的。在第5章我们已经学习过SML指令是由两部分组成的一个4位整数——操作码和操作数。操作码由Simple中的命令决定。例如,Simple命令input对应于SML操作码10(读),Simple命令print对应于SML操作码11(写)。操作数是数据所在的存储单元,这样操作码就可以完成它的任务(例如,操作码10从键盘读取一个值并将它存放在操作数指定的存储单元)。编译器搜索symbolTable来确定每个符号的Simpletron内存位置,这样就可以使用相应的存储单元来完成SML指令。

每个Simple语句都是根据它的命令编译的。例如,在rem语句的行号插入到符号表之后,该语句的其余部分就被编译器忽略,因为注释的目的只是为了说明文档。input语句、print语句、goto语句和end语句分别对应于SML的read、write、branch(到一特定位置)和halt指令。包含这些Simple命令的语句将直接转换到SML(注意,如果goto语句中特定的行号表示Simple程序中其后的语句,那么goto语句就可能包含一个未指定的引用,有时称其为提前引用)。

当对一个未指定引用的goto语句进行编译时,必须对SML指令进行标记,以指示编译器第二步编译必须完成该指令。标记存放在类型为int的有100个元素的flags数组中,其中每个元素都初始化为-1。如果还不知道Simple程序中一个行号指向的存储单元(即它不在符号表中),那么就把该行号存于flags数组的与未完成的指令具有相同下标的元素中。未完成指令的操作数将临时地置为00。例如,一个无条件转移语句(造成一个提前引用)将置为+4000,该值会一直保持到编译器的第二步编译。后面,我们将简要地描述编译器的第二步编译。

if/goto和let语句的编译比其他语句更复杂——它们是产生不止一条SML指令的语句。对一个if/goto语句来说,编译器产生代码来测试条件,如果程序需要,就转移到另一行。转移的结果就是一个未指定的引用。每一个关系运算符和相等运算符都可以通过

SML的 branch zero 和 branch negative 指令（或者可能是二者的联合）来模拟。

对于let语句来说,编译器产生代码可以计算一个任意复杂的包括整型变量和常量的算术表达式,表达式应该用空格将每个运算符和操作数分开,练习 17.12 和练习 17.13 中提供了中缀向后缀转化的算法以及编译器利用后缀表达式来计算表达式值的算法。在运行编译器之前,读者应该完成上面所提到的每个程序。当编译器遇到一个表达式时,它首先将表达式从中缀转换成后缀,然后计算后缀表达式。

编译器是怎样产生机器语言来计算包含变量的一个表达式呢?后缀表达式求值算法有一个“异常指令”(hook),它允许编译器生成SML指令,而不是实际计算表达式。为了在编译器里实现这个hook,必须修改后缀表达式求值算法,以便在符号表中搜索遇到的每个符号(并且可能要将其插入到符号表中);确定符号相应的存储单元,并将该存储单元压入堆栈(代替符号)。如果遇到后缀表达式的一个运算符,那么堆栈顶端的两个存储单元就将弹出,并且通过存储单元作操作数来产生完成该操作的机器语言。每个子表达式的结果将存放在内存的一个临时单元并压入堆栈,这样就能继续计算后缀表达式的值。当后缀表达式计算完成时,存放结果的存储单元就是惟一留在堆栈中的单元。将该结果从堆栈中弹出,然后生成SML指令,从而把结果赋给let语句左边的变量。

## 第二步编译

编译器的第二步编译完成两项任务:确定未指定的引用并向文件输出SML代码。引用的确定如下所示:

- a) 搜索 flags 数组来查找一个未指定的引用(即值不是-1的一个元素)。
- b) 在 symbolTable 数组中定位一个对象,它包含存储在 flags 数组中的符号(确定行号的符号类型是'I')。
- c) 将内存位置从 location 成员中插入到含有未指定引用的指令中(记住,一条含有未指定引用的指令有操作数 00)。
- d) 重复步骤 a)、b)和 c),直到 flags 数组的尾部。

在确定上述过程完成之后,包含SML代码的整个数组将输出到一个磁盘文件中,其中每行都有一条SML指令。Simpletron 读取这个文件并运行(在将模拟程序修改为从文件中得到输入后),编译第一个 Simple 程序产生一个SML文件,然后运行这个文件。

## 一个完整的例子

下面给出了一个完整的由 Simple 编译器完成的把 Simple 程序转换成SML的例子。考虑一个 Simple 程序,它输入一个整数并计算从1到这个整数的和。图 17.25 给出了这个程序和 Simple 编译器第一步编译产生的SML指令,图 17.26 中给出了第一步编译构造的符号表。

Simple 程序	SML 位置和指令	描述
5 rem sum 1 to x	none	忽略 rem
10 input x	00+1099	把 x 读入位置 99
15 rem check y == x	none	忽略 rem
20 if y == x goto 60	01+2098	把 y (98) 装入累加器
	02+3199	从累加器中减去 x (99)
	03+4200	0 分支表示不确定的单元
25 rem increment y	none	忽略 rem

(续表)

simple 程序	SML 位置和指令	描述
30 let y = y + 1	04+2098	把 y 装入累加器
	05+3097	累加器加 1 (97)
	06+2196	存入临时单元 96
	07+2096	从临时单元 96 中读取值
	08+2198	把累加器的值放入 y
35 rem add y to total	none	忽略 rem
40 let t = t + y	09+2095	把 t (95) 装入累加器
	10+3098	累加器加 y
	11+2194	存入临时单元 94
	12+2094	从临时单元 94 中读取值
	13+2195	把累加器的值放入 t
45 rem loop y	none	忽略 rem
50 goto 20	14+4001	跳转到位置 01
55 rem output result	none	忽略 rem
60 print t	15+1195	把 t 的值输出到屏幕上
99 end	16+4300	终止运行

图 17.25 编译器第一步编译后产生的 SML 指令

大多数 Simple 语句都直接转换成单条 SML 指令, 在这个程序中, 注释、第 20 行的 if/goto 语句和 let 语句则例外。注释不需要翻译成机器语言, 但是, 注释语句的行号则放在符号表中, 以防止在 goto 语句和 if/goto 语句中引用这个行号。程序的第 20 行指明: 如果条件  $y==x$  为 true, 则程序控制将转移到第 60 行。因为第 60 行出现在程序后面的位置, 所以编译器的第一步编译不能把 60 放在符号表中 (仅当行号作为语句的第一个标记出现时, 才能将它们放在符号表中)。因此, 这时就不可能确定 SML 的零分支指令的操作数处于 SML 指令数组的 03 位置。编译器把 60 放在 flags 数组的 03 单元, 以指示第二步编译要完成这条指令。

由于 Simple 语句和 SML 指令之间没有一对一的关系, 因此我们必须知道 SML 数组中下一条指令的单元。例如, 第 20 行的 if/goto 语句经编译后产生三条 SML 指令。每次产生一条指令的时候, 我们都必须将指令计数器加 1, 以说明 SML 数组中的下一个单元。如果一个 Simple 程序有很多语句、变量和常量, 那么就要注意 Simpletron 内存大小所带来的一个问题——编译器运行时可能会使内存溢出。为了检测这种情况, 程序中应该有一个数据计数器, 用来记录存放下一个变量或常量的 SML 数组中的单元。如果指令计数器的值大于数据计数器的值, 就表明 SML 数组已经满了。在这种情况下, 编译过程应该终止, 并且编译器应该打印出一条错误信息, 指明在编译过程中产生了内存溢出。这是为了强调尽管程序员完成的编译器没有管理内存的责任, 但是编译器仍应小心确定指令和数据在内存的位置, 并且在编译过程中必须检查诸如内存溢出这样的错误。

符号	类型	位置
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04

(续表)

符号	类型	位置
30	L	04
1	C	97
35	L	09
40	L	09
''	V	95
45	I	14
50	L	14
55	L	15
60	L	15
99	L	16

图 17.26 图 17.25 中程序的符号表

## 逐步查看编译过程

现在让我们追踪图 17.25 所示的 Simple 程序的编译过程。编译器将程序的第一行：

```
5 rem sum 1 to x
```

读入内存。利用 StreamTokenizer 类确定语句的第一个标记（行号）（参看第 8 章中关于 Java 字符串的处理方法）。StreamTokenizer 返回的标记由 Integer.parseInt() 转换成一个整数，这样可以在符号表中定位符号 5。如果这个符号找不到，就把它插入到符号表中。由于我们处在程序的开始位置并且是在第一行，所以表中还没有一个符号。因此，将 5 插入到符号表中，其类型是 L（行号）并将 SML 数组的第一个单元（00）分配给它。尽管这是一行注释，但是在符号表中仍为该行号分配了一个空间（以防止 goto 或 if/goto 引用）。rem 语句不生成 SML 指令，所以指令计数器不会加 1。

接着分解下列语句：

```
10 input x
```

将行号 10 放在符号表中，其类型是 L，并将 SML 数组的第一个单元（00，因为程序以注释开始，所以当前指令计数器为 00）分配给它。命令 input 指明下一个标记是变量（只有变量才出现在 input 语句中）。因为 input 直接对应一个 SML 操作码，因此编译器只需确定 x 在 SML 数组中的单元号。符号表中找不到符号 x，因此代表 x 的 ASCII 码将插入到符号表中，其类型是 V，并在 SML 数组中将单元 99 分配给它（数据存储从 99 开始并向前分配）。现在这条语句能生成 SML 代码，操作码 10 乘以 100，再加上 x 的单元号（由符号表决定）就组成了指令。然后将该指令存放在 SML 数组的 00 单元。因为只生成了一条 SML 指令，所以将指令计数器加 1。

接着标记化下列语句：

```
15 rem check y == x
```

在符号表中搜索行号 15（当然不会找到）。将该行号插入符号表，其类型是 L，并给它分配数组的下一个单元 01（记住，rem 语句不产生代码，所以指令计数器没有加 1）。

下一步分解下列语句：

```
20 if y == x goto 60
```

将行号 20 插入符号表, 其类型是 L, 在 SML 数组中的单元是 01。命令 if 指明将判断一个条件。由于变量 y 在符号表中找不到, 因此将其插入, 并为其赋予类型 V 和 SML 单元 98。下一步将生成 SML 指令来判断该条件。因为 SML 中没有直接的代码与 if/goto 对应, 所以必须利用 x 和 y 完成计算, 并基于计算结果的转移进行模拟。如果 y 等于 x, 即 y 减 x 的结果为 0, 就使用带有计算结果的零分支指令来模拟 if/goto 语句。第一步需要把 y (从 SML 单元 98) 装入累加器。这将产生指令 01 +2098。下一步, 从累加器中减去 x, 这就产生指令 02 +3199。累加器的值可能是 0、正数或负数。由于运算符是 ==, 所以我们希望是零分支。首先, 在符号表中搜索转移位置的内存单元 (本例是 60), 因为没有找到, 所以把 60 放在 flags 数组的 03 单元, 并生成指令 03 +4200 (我们不能加入转移位置的内存单元, 因为还没有在 SML 数组中为行号 60 分配单元)。指令计数器将增加到 04。

编译器接着分析下列语句:

```
25 rem increment y
```

将行号 25 插入到符号表, 它的类型是 L, 并为其分配 SML 单元 04。这时指令计数器不增值。在标记化下列语句时:

```
30 let y = y + 1
```

将行号 30 插入到符号表, 其类型是 L, 并为其分配 SML 单元 04。命令 let 指明该行是一个赋值语句。首先, 这一行的所有符号将插入到符号表 (如果表中找不到它们)。首先将整数 1 将加到符号表中, 它的类型是 C, SML 位置是 97。下一步, 赋值符号的右边从中序转换成后序形式。然后计算后序表达式 (y1+)。在符号表中找出符号 y, 相应的内存单元将压入堆栈。在符号表示中找出符号 1, 并且将相应的内存单元压入堆栈。当遇到运算符 + 时, 运算符右边的操作数和左边的操作数分别弹出堆栈, 然后生成 SML 指令:

```
04 +2098 (装入 y)
05 +3097 (加 1)
```

表达式的结果放入内存的一个临时单元 (96), 相应的指令是:

```
06 +2196 (临时存储)
```

并且临时单元将压入堆栈。现在表达式已经计算完毕, 由于其结果必须放入 y 中 (即 “=” 左边的变量), 所以将临时单元装入累加器并且使用下列指令:

```
07 +2096 (装入临时值)
08 +2198 (存储 y)
```

将累加器存入 y。读者可能会立即注意到, SML 指令显得有些冗余, 我们不久将会讨论这个问题。

当标记化下列语句时:

```
35 rem add y to total
```

将 35 插入符号表, 其类型是 L 并为其分配单元 09。

下列语句:

```
40 let t = t + y
```



类似于30行。变量t将插入到符号表，它的类型是V并为其分配SML单元95。指令的生成与30行具有相同的逻辑和格式，并产生指令09+2095、10+3098、11+2194、12+2094和13+2195。注意，t+y的结果在赋给t(95)之前放在临时单元94。读者会再一次注意到内存单元11和12的指令显得有些冗余。我们会在后面讨论这一点。

下列语句：

```
45 rem loop y
```

是一行注释，因此将行号45加入符号表，其类型是L，SML单元是14。

下列语句：

```
50 goto 20
```

把控制转移到20行。行号50将插入到符号表，类型是L并为其分配SML单元14。goto语句在SML中的相应指令为无条件转移(40)指令，它把控制转移到指定的SML单元。编译器在符号表中搜索行号20，并发现其相应的SML单元01。操作码(40)乘以100并加上单元01产生指令14+4001。

下列语句：

```
55 rem output result
```

是一行注释，因此将55插入符号表，其类型是L，SML单元是15。下列语句：

```
60 print t
```

是一条输出语句。行号60将插入到符号表，其类型是L，并为其分配SML单元15。print在SML中的相应操作码为11(写)。由符号表确定的t的单元与操作码乘以100的结果相加，这样就生成了指令。

下列语句：

```
99 end
```

是程序的最后一行。行号99将存储在符号表中，它的类型是L，并为其分配SML单元16。end命令产生SML指令+4300(43是SML中的停机码)，它将作为最后一条指令写入SML内存数组。

这样就完成了编译器的第一步编译。现在我们考虑第二步编译。搜索flags数组中不等于-1的值，我们找到单元03，它存放的数值是60，因此编译器知道指令03尚未完成。通过在符号表中搜索数值60，继而确定它的单元并将该单元加到未完成的指令中，随后编译器完成了这条指令。在这种情况下，通过搜索能确定60行对应于SML单元15，因此编译器产生完成的指令03+4215来代替03+4200。现在，已经成功地编译了这个Simple程序。

为了建立这个编译器，必须完成下述的每个任务：

- a) 修改练习5.43中编写的Simpletron模拟程序，以便让它从用户指定的文件中得到输入(请参见第15章)。该模拟程序应把它的结果以和屏幕输出相同的格式写到一个磁盘文件中。把该模拟程序转换成面向对象的程序，特别是把硬件的每个部分作为一个对象。利用继承把指令类型安排成类层次的形式。然后，使用executeInstruction消息通知每条指令让其运行，这样程序就能多态地运行

- b) 修改练习 17.12 的中序向后序转换的算法, 使它能处理多位整数的操作数和用单个字母命名的变量操作数。提示: StreamTokenizer 类能在一个表达式中定位每个常量和变量, 而 Integer 类中的 parseInt 方法能将常量从字符串转换成整数 (注意: 必须改变后序表达式中的数据表示, 以便支持变量名和整数常量)。
- c) 修改计算后序表达式的算法, 使它能够处理多位整数的操作数和变量名操作数。该算法也必须实现上面讨论的 hook 功能, 以便不用直接计算表达式就能产生 SML 指令。提示: StreamTokenizer 类能在一个表达式中定位每个常量和变量, 而 Integer 类中的 parseInt 方法能把常量从字符串转换成整数 (注意: 必须改变后序表达式中的数据表示, 以便支持变量名和整数常量)。
- c) 建立编译器。合并 b) 和 c) 来计算 let 语句中的表达式。编写完成的程序应该既包括完成第一步编译的方法, 也包括完成编译器的第二步编译的方法, 这两个方法都能调用其他的方法来完成它们的任务。尽可能地让你的编译器采用面向对象的技术。
- 17.28 (优化 Simple 编译器) 一个程序经过编译并转换成 SML 后, 就会产生一系列指令。指令间的某些组合经常重复出现 (通常是三个一组), 这样的一组指令称为“结果生成指令” (production)。正常情况下, 一个结果生成指令由装入、相加和存储这三条指令组成。例如, 图 17.27 给出了图 17.25 的程序编译过程中产生的五条指令。前三条指令是结果生成指令 y 加上 1。注意, 指令 06 和 07 把累加器的值存储在临时单元 96, 然后再把值放回累加器, 使得指令 08 能把值存储在单元 98 中。通常, 一条装入指令会跟在一个结果生成指令之后, 它对刚存储过的单元进行操作。通过取消对相同内存单元操作的存储指令和随后的装入指令, 就可以优化代码, 这样 Simpletron 就能够更快地执行程序。图 17.28 给出了图 17.25 中的程序经过优化后的代码。注意, 优化代码中少了 4 条指令——内存空间节省了 25%。

```

1 04 +2098 (装入)
2 05 +3097 (相加)
3 06 +2196 (存储)
4 07 +2096 (装入)
5 08 +2198 (存储)

```

图 17.27 图 17.25 中程序的未优化代码

Simple 程序	SML 单元和指令	描述
5 rem sum 1 to x	none	忽略 rem
10 input x	00+1099	把 x 读入单元 99
15 rem check y == x	none	忽略 rem
20 if y == x goto 60	01+2098	把 y (98) 装入累加器
	02+3199	从累加器中减去 x (99)
	03+4211	结果为 0 则转到单元 11
25 rem increment y	none	忽略 rem
30 let y = y + 1	04+2098	把 y 装入累加器
	05+3097	累加器加 1 (97)
	06+2198	把累加器的值放入 y (98)
35 rem add y to total	none	忽略 rem
40 let t = t + y	07+2096	从单元 96 装入 t
	08+3098	累加器加 y (98)
	09+2196	把累加器的值放入 t (96)
45 rem loop y	none	忽略 rem

(续表)

simple 程序	SML 单元和指令	描述
50 goto 20	10+4001	跳转到单元 01
55 rem output result	none	忽略 rem
60 print t	11+1196	把 t (96) 的值输出到屏幕上
99 end	12+4300	终止运行

图 17.28 图 17.25 中程序的优化代码

修改编译器,使它提供一个能够优化其生成的 Simpletron 机器语言的选项。手工比较未优化的代码和优化后的代码,并计算减少的百分比。

17.29 (修改 Simple 编译器)对 Simple 编译器完成下述修改。其中一些修改也需要对练习 5.43 中的 Simpletron 模拟程序进行修改。

- 允许 let 语句中使用取模运算符 (%)。这里必须修改 Simpletron 机器语言,使它包括一个取模指令。
- 在 let 语句中允许求幂运算,将 “^” 作为求幂运算符。这里必须修改 Simpletron 机器语言,使它包括一个求幂指令。
- 在 Simple 语句中允许编译器识别大小写字母 (例如 'A' 等价于 'a'),这里不需要修改 Simpletron 模拟程序。
- 允许 input 语句像 “input x, y” 一样读取多个变量的值,这里不需要修改 Simpletron 模拟程序。
- 允许编译器像 “print a, b, c” 一样在一条 print 语句中输出多个值,这里不需要修改 Simpletron 模拟程序。
- 在编译器中加入语法检查功能,使得在 Simple 程序中遇到语法错误时能够输出错误信息,这里不需要修改 Simpletron 模拟程序。
- 允许整数数组,这里不需要修改 Simpletron 模拟程序。
- 允许使用 Simple 命令 gosub 和 return 指定的子例程。命令 gosub 将程序控制送入一个子例程,命令 return 将控制返回到 gosub 之后的语句,这类似于 Java 中的方法调用。程序中分布的许多 gosub 命令能调用同一个子例程,这里不需要修改 Simpletron 模拟程序。
- 允许如下形式的重复结构:

```
for x = 2 to 10 step 2
    Simple 语句
next
```

这个 for 语句从 2 循环到 10,增量是 2, next 行表示循环体的结束,这里不需要修改 Simpletron 模拟程序。

- 允许如下形式的重复结构:

```
for x = 2 to 10
    Simple 语句
next
```

这个 for 语句从 2 循环到 10,增量是 1, next 行表示循环体的结束,这里不需要修改 Simpletron 模拟程序。

- k) 允许编译器处理字符串的输入和输出。这需要修改 Simpletron 模拟程序以处理字符串的存储。提示：每个 Simpletron 字都将分为两部分，每部分都包括一个两位整数。每个两位整数代表一个字符的十进制 ASCII 码。增加一条机器语言指令，让它打印从某个 Simpletron 内存单元开始的字符串。这个单元的字的后半部分是该字符串的字符数目的计数（即字符串的长度）。其后的每半字包含用两位十进制数表示的单个字符的 ASCII 码。这条机器语言指令检查字符串的长度，然后把每个两位数字转化成等价的字符串并打印出来。
- l) 允许编译器能够处理除了整数之外的浮点数，这里也必须修改 Simpletron 模拟程序以处理浮点数

**17.30 (一个简单的解释器)** 解释器就是一个程序，它读入使用高级语言编写的程序语句，确定语句所执行的操作，并立即执行该操作。这个高级语言程序并不首先转换成机器语言。解释器比编译器执行的速度慢，因为执行时必须首先解释程序中遇到的每条需要解释的语句。如果语句包含在循环中，那么在循环中每次遇到这些语句时都必须解释它们。Basic 编程语言的早期版本就是用解释器实现的。大部分 Java 程序也是通过解释过程而运行的。

为练习 17.26 讨论的 Simple 语言编写一个解释器。该程序应该使用在练习 17.12 中开发的中序向后序转换的程序，以及在练习 17.13 中开发的计算后序表达式的程序，从而可以计算 let 语句中的表达式。本程序的限制与练习 17.26 的 Simple 语言中的限制相同。把解释器运行这些程序的结果同 Simpletron 模拟程序（参见练习 15.12）中编译执行的结果进行比较。

**17.31 (在链表的任何位置进行插入/删除)** 我们的链表类只允许用户在链表的头部和尾部进行插入和删除操作。当我们利用继承或复合来重用链表类，从而使用最少的代码来产生一个堆栈类和队列类时，这些功能是很方便的。正常情况下，链表比我们提供的那些类更加通用化。修改我们在本章开发的链表类，完成在链表的任何单元进行插入和删除的操作。

**17.32 (没有尾指针的链表和队列)** 我们实现的链表（参见图 17.3）使用了 firstNode 和 lastNode。对于链表类的 insertAtBack 方法和 removeFromBack 方法来说，lastNode 是很有用的。insertAtBack 方法相当于 Queue 类的 enqueue 方法。重写链表类使它不使用 lastNode。这样，在链表末尾的任何操作都要从表头开始查找。这会影响我们 Queue 类的实现吗（参见图 17.12）？

**17.33 (二叉树的排序和查找)** 进行二叉树排序的问题是数据的插入顺序会影响树的形状——对于相同的数据集合，不同的插入顺序会生成不同形状的二叉树。二叉树的排序和查找算法的性能与二叉树形状的关系很大。如果数据以升序或降序插入，那么该二叉树会是什么形状呢？如果要获得最佳的查找性能，那么二叉树应该是什么形状呢？

**17.34 (索引表)** 正如本章所提到的，应该顺序地查找链表。对于大型链表，这样查找的性能很差。改善链表查找性能的常用技术是创建和维护链表的一个索引。索引是指向链表中不同关键单元的一系列指针。例如，当搜索一个大型的名字链表时，可以创建一个有 26 个项的索引——每一项对应字母表的一个字母，这样就能提高性能。如果要查找以“Y”开头的姓氏，首先要通过索引找到“Y”项开始的单元，然后“跳到”表中的这个位置，并进行线性查找直到找到所需的姓氏为止。利用图 17.3 所示的 List 类作为 IndexList 类的基础。编写一个示例索引表操作的程序，一定要包括 insertIndexedList 方法、searchIndexedList 方法和 deleteFromIndexedList 方法。

## 第 18 章 Java 工具包和位处理

### 教学目标

- 理解诸如 Vector 和 Stack 类这样的容器以及 Enumeration 接口
- 学会创建 Hashtable 对象和称为 Properties 对象的永久的散列表
- 使用位处理和 BitSet 对象
- 建立一个流行的具有 Observable 对象和 Observer 接口的 OO 设计模式

### 18.1 简介

这一章我们将讨论 java.util 软件包中大量的工具类。我们考查了 Vector 类，它使我们能够创建类似于数组的对象，这些对象在程序数据的存储需求改变时能够动态地扩大和缩小。我们也将考虑 Enumeration 接口，该接口和 Vector 类一起使用，以允许一个程序重复地使用诸如 Vector 这样的容器中的元素。

我们将讨论 Stack 类，它提供传统的堆栈操作 push 和 pop，以及我们在第 17 章中没有考虑过的其他一些操作。

我们还会介绍抽象类 Dictionary，它为我们提供了使用表存储数据和检索这些数据时所使用的框架。我们将解释“散列”（hashing）理论，一种可以在表中快速存储和检索数据的技术，并且还将示范如何使用 Java 中的 Hashtable 类来建立和处理散列表。我们将考虑的 Properties 类，它使我们能够创建永久的散列表，即能通过输出流写到一个文件中，并能按照要求从输入流将其读回到系统中的散列表。

为了处理日期和时间，我们复习一下 Date 类。这一章将概述 Random 类，与 Math.random 类相比，该类能提供更丰富的随机数功能。

我们将在面向对象的系统中讨论一种流行的设计模式，即当第一个对象的状态改变时，由一个对象通知其他对象；我们使用 Observable 类和 Observer 接口来实现这种模式。

我们还要对位处理运算符进行更为广泛的讨论，然后讨论 BitSet 类，后者能建立类似于位数组的对象并得到单个的位数组。

### 18.2 Vector 类和 Enumeration 接口

在大多数编程语言（包括 Java）中，传统的数组具有固定长度——它们不能根据应用程序中存储需求的改变而扩大和缩小。Java 的 Vector 类提供类数组的数据结构，它能动态地改变自身的大小。

任何时候，Vector 都包括少于或等于它的容量的一定数量的元素。这个容量是为数组预先保留的。

**性能提示 18.1**

将一个新的数据插入目前规模小于它的容量的一个数组，相对而言是比较快的操作。

如果需要增大 Vector，那么它可以按照系统默认或用户的要求增大。

**性能提示 18.2**

如果在数组中插入一个新元素的同时需要增加它的大小以适应新的要求，那么这将是一个相对耗时的操作

如果不指明数组需要扩大的容量，则系统会自动将 Vector 的现有容量大小增加一倍。

**性能提示 18.3**

按默认值一次将 Vector 容量增加一倍的方法看起来有些浪费系统空间，但是对于许多快速增加的 Vector，这样做恰好是能够“接近要求”的有效方法。与每次增加一个元素就使 Vector 容量进行相应的增长相比，这样操作的时间效率更高，尽管同时有可能浪费系统空间。

**性能提示 18.4**

如果存储空间非常宝贵，那么可以使用 Vector 类的方法 `trimToSize` 来按照数组实际大小调整数组容量。这将优化 Vector 的存储空间，但使用时请小心谨慎。如果需要在 Vector 中插入一个元素，那么程序运行速度将变得很慢，因为它强制 Vector 按照动态需求进行增长——这就使得数组中没有为扩充预留的空间。

Vector 设计用来存放 Object 的引用。也就是任何类型的对象的引用都可以存储在一个 Vector 中。如果要在 Vector 中存储原始数据类型的值，那么就必须使用相应类型的包装类 (Integer、Long、Float、Double、Boolean 和 Character)，它们放在 `java.lang` 软件包中，并用来创建包含原始数据类型值的对象。

图 18.1 中的 applet 演示了 Vector 类及其方法，每个方法对应一个按钮。用户可以在提供的文本字段中键入一个 String，然后点击各个按钮，程序将在 applet 的状态区域中显示相应的信息，以解释每个操作的结果。

这个 applet 的方法 `init` 采用下面的语句创建了一个 Vector：

```
Vector v = new Vector(1);
```

上述语句创建了一个初始容量为 1 的 Vector。这个 Vector 每次在需要增加大小时都将自动把容量增长一倍。

```
1 // Fig. 18.1: VectorTest.java
2 // Testing the Vector class of the java.util package
3 import java.util.*;
4 import java.awt.*;
5 import java.applet.Applet;
6
7 public class VectorTest extends Applet {
8     Vector v;
9
10    // GUI components
11    Label prompt;
12    TextField input;
13    Button addBtn, removeBtn, firstBtn, lastBtn,
14        emptyBtn, containsBtn, locationBtn,
15        trimBtn, statsBtn, displayBtn;
16}
```

```
17     public void init()
18     {
19         v = new Vector( 1 );
20
21         prompt = new Label( "Enter a string" );
22         input = new TextField( 10 );
23         addBtn = new Button( "Add" );
24         removeBtn = new Button( "Remove" );
25         firstBtn = new Button( "First" );
26         lastBtn = new Button( "Last" );
27         emptyBtn = new Button( "Is Empty?" );
28         containsBtn = new Button( "Contains" );
29         locationBtn = new Button( "Location" );
30         trimBtn = new Button( "Trim" );
31         statsBtn = new Button( "Statistics" );
32         displayBtn = new Button( "Display" );
33
34         add( prompt );
35         add( input );           // value to add, remove or locate
36         add( addBtn );         // add the input value
37         add( removeBtn );     // remove the input value
38         add( firstBtn );      // look at the first element
39         add( lastBtn );       // look at the last element
40         add( emptyBtn );      // check if stack is empty
41         add( containsBtn );   // does vector contain input value?
42         add( locationBtn );   // location of input value
43         add( trimBtn );       // trim vector to number of elements
44         add( statsBtn );      // display statistics
45         add( displayBtn );    // display the stack contents
46     }
47
48     public boolean action( Event e, Object o )
49     {
50         if ( e.target == addBtn ) {
51             v.addElement( input.getText() );
52             showStatus( "Added to end: " + input.getText() );
53         }
54         else if ( e.target == removeBtn ) {
55             if ( v.removeElement( input.getText() ) )
56                 showStatus( "Removed: " + input.getText() );
57             else
58                 showStatus( input.getText() + " not in vector" );
59         }
60         else if ( e.target == firstBtn ) {
61             try {
62                 showStatus( "First element: " + v.firstElement() );
63             }
64             catch ( NoSuchElementException exception ) {
65                 showStatus( exception.toString() );
66             }
67         }
68         else if ( e.target == lastBtn ) {
69             try {
70                 showStatus( "Last element: " + v.lastElement() );
71             }
72             catch ( NoSuchElementException exception ) {
73                 showStatus( exception.toString() );
74             }
75         }
76         else if ( e.target == emptyBtn ) {
77             showStatus( v.isEmpty() ? "Vector is empty" :
```

```

78             "Vector is not empty" );
79         }
80         else if ( e.target == containsBtn ) {
81             String searchKey = input.getText();
82
83             if ( v.contains( searchKey ) )
84                 showStatus( "Vector contains " + searchKey );
85             else
86                 showStatus( "Vector does not contain " + searchKey );
87         }
88         else if ( e.target == locationBtn ) {
89             showStatus( "Element is at location " +
90                 v.indexOf( input.getText() ) );
91         }
92         else if ( e.target == trimBtn ) {
93             v.trimToSize();
94             showStatus( "Vector trimmed to size" );
95         }
96         else if ( e.target == statsBtn ) {
97             showStatus( "Size = " + v.size() +
98                 "; capacity = " + v.capacity() );
99         }
100        else if ( e.target == displayBtn ) {
101            Enumeration enum = v.elements();
102            StringBuffer buf = new StringBuffer();
103
104            while ( enum.hasMoreElements() )
105                buf.append( enum.nextElement() ).append( " " );
106
107            showStatus( buf.toString() );
108        }
109
110        input.setText( "" );
111        return true;
112    }
113 }

```

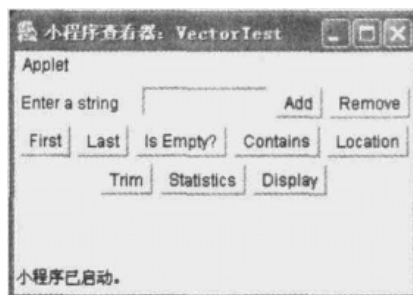


图 18.1 演示 java.util 软件包中的 Vector 类

Vector 类提供了两种其他的构造函数。没有参数的构造函数按照初始容量 10 来创建一个空的 Vector；带有两个参数的构造函数把第一个参数的值作为初始容量来创建一个 Vector，第二个参数作为容量的增量。每次当这个 Vector 需要增长时，都将按照这个值来增加数组的容量。

下列表达式：

```
v.addElement( input.getText() )
```

使用了 Vector 的方法 addElement 来将它的参数增加在 Vector 的尾部。这个 Vector 的容量将自动增长（如果需要），以便存储新的元素。Vector 类还提供了 insertElementAt 方法，用来将一个元素（这个



方法的第一个参数)插入到 Vector 的一个指定位置上(方法的第二个参数); setElementAt 方法将元素(这个方法的第一个参数)放在 Vector 的一个指定位置上(方法的第二个参数)。insertElementAt 方法通过移动元素而自动增加空间。setElementAt 方法将用它的参数替换指定位置上的元素值。

下列表达式:

```
v.removeElement( input.getText() )
```

使用 Vector 的 removeElement 方法来将与其参数匹配的的第一个元素从 Vector 中删除。如果在 Vector 中发现这个元素,则该方法返回 true; 否则将返回 false。如果删除该元素,所有这个元素之后的数组成员将向前移动一个位置,从而填补这个删除的空间。Vector 类同时还提供 removeAllElement 方法来将 Vector 中的所有元素删除, removeElementAt 方法将删除由参数指定的位置上的元素。

下列表达式:

```
v.firstElement( input.getText() )
```

使用 Vector 的 firstElement 方法返回 Vector 的第一个元素的引用。如果当前在 Vector 中没有任何元素,这个方法将抛出一个 NoSuchElementException 异常。

下列表达式:

```
v.lastElement()
```

使用 Vector 的 lastElement 方法返回 Vector 最后一个元素的引用。如果当前在 Vector 中没有任何元素,这个方法将抛出一个 NoSuchElementException 异常。

下列表达式:

```
v.isEmpty()
```

使用 Vector 的方法 isEmpty 来判断 Vector 是否为空。如果 Vector 中没有任何元素,那么这个方法将返回 true; 否则返回 false。

下列表达式:

```
v.contains( searchKey )
```

使用 Vector 的 contains 方法来判断 Vector 是否包含作为参数所指定的 searchKey。如果 searchKey 出现在 Vector 中,这个方法将返回 true; 否则返回 false。contains 方法使用 Object 的 equals 方法来判断 searchKey 是否等于 Vector 的某一个元素。

下列表达式:

```
v.indexOf( input.getText() )
```

使用 Vector 的 indexOf 方法来取得 Vector 中参数第一个位置的下标。如果参数在 Vector 中没有找到,这个方法将返回 -1。这个方法还有另一个版本,它把第二个参数指定为从 Vector 开始搜寻的下标。

下列表达式:

```
v.trimToSize()
```

使用 Vector 的 trimToSize 方法来调整 Vector 的容量,使其等于当前 Vector 中元素的确切数目(即 Vector 的大小)。

下列表达式:

```
v.size()  
v.capacity()
```

使用 `Vector` 的 `size` 方法和 `capacity` 方法来分别判断当前 `Vector` 中元素的个数，以及在不需重新分配空间的前提下，能够存储在 `Vector` 中的元素个数。

下列表达式：

```
Enumeration enum = v.elements()
```

使用 `Vector` 的 `elements` 方法返回包含 `Vector` 中元素的一个 `Enumeration` 的引用。`Enumeration` 提供了两个用来一次遍历一个元素集合的方法。下列表达式：

```
enum.hasMoreElements();
```

返回 `true`，直到在 `Enumeration` 中不再包含元素。下列表达式：

```
enum.nextElement();
```

返回 `Enumeration` 中下一个元素的引用。如果下面没有元素，则将抛出一个 `NoSuchElementException` 异常。

如果想了解更多 `Vector` 方法的信息，可以查看 Java API 的说明文档。

## 18.3 Stack 类

在第 17 章中，我们学习了如何建立诸如链表、堆栈、队列和树这样的基本数据结构。在 Java 中我们通常是“重用、重用再重用”，而不再是编写每一种需要的数据结构，所以我们常常要学会利用现有的数据结构类。在本节中，我们研究 Java 工具包 (`java.util`) 中的 `Stack` 类。

我们已经讨论过 `Vector` 类，它实现了一个动态大小的数组。`Stack` 类扩展了 `Vector` 类，以实现一个堆栈类型的数据结构。如同 `Vector` 一样，`Stack` 类设计用来存储 `Object`；为了存储原始数据类型，必须使用适当的包装类 (`Boolean`、`Character`、`Integer`、`Long`、`Float` 或 `Double`)。

图 18.2 中的 applet 提供了一个图形用户界面，通过它能够测试 `Stack` 中的每一个方法。下列表达式：

```
s = new Stack()
```

创建了一个空的 `Stack`。

```
1 // Fig. 18.2: StackTest.java  
2 // Testing the Stack class of the java.util package  
3 import java.util.*;  
4 import java.awt.*;  
5 import java.applet.Applet;  
6  
7 public class StackTest extends Applet {  
8     Stack s;  
9  
10    // GUI components  
11    Label prompt;  
12    TextField input;  
13    Button pushBtn, popBtn, peekBtn,  
14        emptyBtn, searchBtn, displayBtn;
```

```
15
16     public void init()
17     {
18         s = new Stack();
19
20         prompt = new Label( "Enter a string" );
21         input = new TextField( 10 );
22         pushBtn = new Button( "Push" );
23         popBtn = new Button( "Pop" );
24         peekBtn = new Button( "Peek" );
25         emptyBtn = new Button( "Is Empty?" );
26         searchBtn = new Button( "Search" );
27         displayBtn = new Button( "Display" );
28         add( prompt );
29         add( input );          // value to push or search for
30         add( pushBtn );        // push the input value
31         add( popBtn );         // pop a value
32         add( peekBtn );        // peek at the top
33         add( emptyBtn );       // check if stack is empty
34         add( searchBtn );      // search for input value
35         add( displayBtn );     // display the stack contents
36     }
37
38     public boolean action( Event e, Object o )
39     {
40         if ( e.target == pushBtn ) {
41             showStatus( "Pushed: " + s.push( input.getText() ) );
42         }
43         else if ( e.target == popBtn ) {
44             try {
45                 showStatus( "Popped: " + s.pop() );
46             }
47             catch ( EmptyStackException exception ) {
48                 showStatus( exception.toString() );
49             }
50         }
51         else if ( e.target == peekBtn ) {
52             try {
53                 showStatus( "Top: " + s.peek() );
54             }
55             catch ( EmptyStackException exception ) {
56                 showStatus( exception.toString() );
57             }
58         }
59         else if ( e.target == emptyBtn ) {
60             showStatus( s.empty() ? "Stack is empty" :
61                         "Stack is not empty" );
62         }
63         else if ( e.target == searchBtn ) {
64             String searchKey = input.getText();
65             int result = s.search( searchKey );
66
67             if ( result == -1 )
68                 showStatus( searchKey + " not found" );
69             else
70                 showStatus( searchKey + " found at element " +
```

```
71             result );
72     }
73     else if ( e.target == displayBtn ) {
74         Enumeration enum = s.elements();
75         StringBuffer buf = new StringBuffer();
76
77         while ( enum.hasMoreElements() )
78             buf.append( enum.nextElement() ).append( " " );
79
80         showStatus( buf.toString() );
81     }
82
83     return true;
84 }
85 }
```

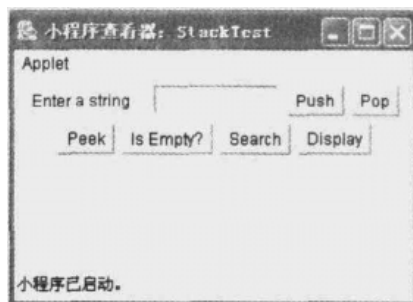


图 18.2 演示 java.util 软件包中的 Stack 类

下列表达式:

```
s.push( input.getText() )
```

使用 Stack 的 push 方法来将其参数加到堆栈的顶端。该方法返回一个 Object, 指向 push 的参数。

下列表达式:

```
s.pop()
```

使用 Stack 的 pop 方法移出栈顶元素。该方法返回一个 Object, 指向移出的元素。如果 Stack 内没有元素, 就将抛出 EmptyStackException 异常。

下列表达式:

```
s.peek()
```

使用 Stack 的 peek 方法在不移出栈顶元素的前提下寻找栈顶元素。这个方法返回指向该元素的一个 Object。

下列表达式:

```
s.empty()
```

使用 Stack 的 empty 方法来判断堆栈是否为空。如果是, 则返回 true; 否则返回 false。

下列表达式:

```
int result = s.search( searchKey );
```

使用Stack的search方法来判断它的参数是否在堆栈内。如果在,则返回该元素在堆栈内的位置。栈顶元素在位置1。如果该元素不在堆栈内,则返回-1。

请注意,Vector类全部的公有接口实际上是Stack类的一部分,因为Stack类是从Vector类继承而来的。为了证实这一点,我们的例子提供了一个可以显示栈中内容的按钮。该按钮调用相应的方法来得到包括栈中内容的Enumeration。然后,该Enumeration用来遍历栈中的元素。

#### 测试与调试提示 18.1

因为Stack类从Vector类继承而来,所以用户可以对Stack的对象执行一些通常不能对传统堆栈数据结构进行的操作。这会影响Stack的元素并破坏Stack的完整性。

## 18.4 Dictionary 类

Dictionary主要用于将关键字转换成值,该类接收一个关键字并返回一个值。Dictionary是一个抽象类,它是我们将要讨论的Hashtable类的一个超类。Dictionary类提供用来维护由“关键字/值”组成的表所必需的一些公有接口的方法。在该表中,关键字为表中的“行”而值是表中的“列”。

size方法返回Dictionary对象所占有的关键字/值对的数目。如果Dictionary是空的,则isEmpty方法返回true,否则返回false。keys方法返回一个Enumeration,其中给出了Dictionary中的关键字;elements方法给出了Dictionary的元素;get方法返回一个与给定关键字的值对应的对象;remove方法删除了一个对应于给定关键字的元素并返回指向它的指针。

## 18.5 Hashtable 类

在面向对象的编程语言中,创建一个新类型是很容易的。如果一个程序创建了新的或已有类型的对象,那么就需要有效地管理这些对象,其中包括存储和检索对象。如果数据的某些方面与关键字/值直接匹配,并且这些关键字都是惟一的和紧密相连的,那么采用数组存储和检索信息就比较有效。如果有100个雇员并且每个人都有9位的社会保险号,然后利用社会保险号作为关键字来存储和检索雇员数据,那么表面上看就需要一个有999 999 999个元素的数组,这是因为共有999 999 999个独一无二的9位编号。实际上对于所有应用程序来说,将社会保险号作为关键字是不切实际的。但是,如果可以使用那么大的数组,那么通过简单地使用社会保险号作为数组索引,就可以非常高效地存储和检索雇员记录。

许多应用程序都有这样的问题:可能是关键字的类型产生了错误(不是非负整数);或者它们的类型虽然正确,但是却稀疏地分布在很大的范围中。

我们需要一个高速的方案,以把社会保险号、零件库存号码以及类似的关键字转换成惟一的数组下标。如果一个应用程序需要存储某些内容,该方案就能迅速地把应用程序的关键字转换成下标并把信息记录存放在数组的关键字/值对中。数据检索是这样完成的:一旦应用程序得到想要检索的数据记录的关键字的值,就简单地对关键字应用转换方案,从而产生存储在数组中的数据的下标,然后就可以检索出该数据。

上面所描述的方案就是散列(hashing)技术的基础。为什么使用这样一个名字呢?因为当我们把关键字转换成数组下标时,是一位位地拼凑成一种“混杂”的编号。该编号并没有真正的实际意义,它只在存储和检索使用该编号组织的数据记录时有用。

在这种方案中,如果产生了系统冲突,就会发生误操作,也就是两个不同的关键字“散列”在

数组中的同一个单元（或元素）。因为我们不能把两条不同的数据记录存储在同一个位置，所以需要为所有散列在同一数组下标的所有记录（第一个除外）找到一个不同于该下标的替换位置。许多方案都可以实现这一点。其中一种称为“再次散列”，即重复对关键字进行散列转换，以得到数组中的下一个候选单元。该散列过程设计成完全随机的，因此必然会出现只有少部分数据能够找到一个散列出来的可用单元的情况。

另一种方案是只使用一次散列来找到第一个候选单元。如果该单元不为空，那么将线性地搜索后继单元，直到找到可用的单元为止。检索工作采用同样的方法：散列一次关键字，检查所指向的单元是否包含所需要的信息。如果是，则搜索结束；否则就线性地搜索后继单元直到找到所要的数据。

最常用的解决散列表冲突的方法是让表中的每个单元都有一个散列“桶”，一般是散列到该单元的所有关键字/值对所组成的链表。Java 的 `Hashtable` 类（在 `java.util` 软件包中）就是采用这种方法实现的。

影响散列方案实现的一个因素称为“负荷因子”（load factor）。它是散列表中已占用的单元数目与散列表大小的比值。这个比值越接近 1.0，冲突的机会就越大。

#### 性能提示 18.5

散列表的负荷因子是衡量以时间换空间的最好指标：通过增加负荷因子，我们可以更好地利用内存，但由于增加了散列冲突，程序的运行速度就会变慢。降低负荷因子，由于减少了散列冲突，程序的运行速度会加快，但我们内存就利用得不充分，因为散列表中有较大一部分是空的。

对于许多不经常编程的人来说，恰当地使用散列表是一件很复杂的工作。计算机专业的学生在“数据结构”课程上会学习很多有关散列方案的知识。由于散列值困扰着许多程序员，因此 Java 提供了 `Hashtable` 类和一些相关的属性，以帮助程序员在不考虑细节的情况下使用散列。

实际上，在我们学习面向对象编程的过程中，上面的这些内容是非常重要的。类封装隐藏了复杂性（即实现细节），并提供了友好的用户界面。在面向对象的编程领域中，恰当地编写类来实现这一点是最有价值的技巧。

图 18.3 中的 applet 提供了一个图形用户界面，使用它能够测试 `Hashtable` 的每一个方法。下列语句：

```
table = new Hashtable( );
```

创建了一个空的 `Hashtable`，它的默认容量是 101，默认负荷因子是 0.75。当散列表中占用的关键字/值对的数目超过了容量与负荷因子的乘积时，该表就会自动增大。`Hashtable` 类也分别提供一个将容量作为参数的构造函数，以及将容量和负荷因子作为参数的构造函数。

下列语句：

```
object val = table.put( lName.getText(), emp );
```

使用 `Hashtable` 的 `put` 方法来向 `Hashtable` 中增加一个关键字（第一个参数）和一个值（第二个参数）。如果 `Hashtable` 中没有指定关键字的值，就返回 `null`。如果 `Hashtable` 表中存在指定关键字的值，那么就返回 `Hashtable` 中的原始值，这样可以帮助程序处理准备替换指定关键字/值的情况。如果关键字和值有一个为 `null`，就会抛出 `NullPointerException` 异常。下列语句：

```
object val = table.get( lName.getText() )
```

能够在 `Hashtable` 中找到值，它使用 `Hashtable` 的方法 `get` 来找到与作为参数的关键字相对应的值。如果存在值，该方法就返回指向该值的一个 `Object`；否则返回 `null`。

```
1 // Fig. 18.3: HashtableTest.java
2 // Demonstrates class Hashtable of the java.util package.
3 import java.util.*;
4 import java.awt.*;
5 import java.applet.Applet;
6
7 public class HashtableTest extends Applet {
8     Hashtable table;
9
10    Label name1, name2;
11    TextField fName, lName;
12    TextArea display;
13    Button put, get, remove, empty, containsKey, containsObj,
14        clear, listElems, listKeys;
15
16    public void init()
17    {
18        table = new Hashtable();
19
20        name1 = new Label( "First name" );
21        fName = new TextField( 10 );
22        name2 = new Label( "Last name (key)" );
23        lName = new TextField( 10 );
24        display = new TextArea( 4, 45 );
25        put = new Button( "Put" );
26        get = new Button( "Get" );
27        remove = new Button( "Remove" );
28        empty = new Button( "Empty" );
29        containsKey = new Button( "Contains key" );
30        containsObj = new Button( "Contains object" );
31        clear = new Button( "Clear table" );
32        listElems = new Button( "List objects" );
33        listKeys = new Button( "List keys" );
34        add( name1 );
35        add( fName );
36        add( name2 );
37        add( lName );
38        add( display );
39        add( put );
40        add( get );
41        add( remove );
42        add( empty );
43        add( containsKey );
44        add( containsObj );
45        add( clear );
46        add( listElems );
47        add( listKeys );
48    }
49
50    public boolean action( Event e, Object o )
51    {
52        if ( e.target == put ) {
53            Employee emp = new Employee( fName.getText(),
54   lName.getText() );
55            Object val = table.put( lName.getText(), emp );
56        }
```

```
57         if ( val == null )
58             showStatus( "Put: " + emp.toString() );
59         else
60             showStatus( "Put: " + emp.toString() +
61                         "; Replaced: " + val.toString() );
62     }
63     else if ( e.target == get ) {
64         Object val = table.get( lName.getText() );
65
66         if ( val != null )
67             showStatus( "Get: " + val.toString() );
68         else
69             showStatus( "Get: " + lName.getText() +
70                         " not in table" );
71     }
72     else if ( e.target == remove ) {
73         Object val = table.remove( lName.getText() );
74
75         if ( val != null )
76             showStatus( "Remove: " + val.toString() );
77         else
78             showStatus( "Remove: " + lName.getText() +
79                         " not in table" );
80     }
81     else if ( e.target == empty ) {
82         showStatus( "Empty: " + table.isEmpty() );
83     }
84     else if ( e.target == containsKey ) {
85         showStatus( "Contains key: " +
86                     table.containsKey( lName.getText() ) );
87     }
88     else if ( e.target == containsObj ) {
89         showStatus( "Contains object: " +
90                     table.contains( new Employee( fName.getText(),
91   lName.getText() ) ) );
92     }
93     else if ( e.target == clear ) {
94         table.clear();
95         showStatus( "Clear: Table is now empty" );
96     }
97     else if ( e.target == listElems ) {
98         StringBuffer buf = new StringBuffer();
99
100         for ( Enumeration enum = table.elements();
101              enum.hasMoreElements(); )
102             buf.append( enum.nextElement() ).append( '\n' );
103
104         display.setText( buf.toString() );
105     }
106     else if ( e.target == listKeys ) {
107         StringBuffer buf = new StringBuffer();
108
109         for ( Enumeration enum = table.keys();
110              enum.hasMoreElements(); )
111             buf.append( enum.nextElement() ).append( '\n' );
112     }
```



```

113         display.setText( buf.toString() );
114     }
115
116     return true;
117 }
118 }
119
120 class Employee {
121     private String first;
122     private String last;
123
124     public Employee( String fName, String lName )
125     {
126         first = fName;
127         last = lName;
128     }
129
130     public String toString() { return first + " " + last; }
131 }

```

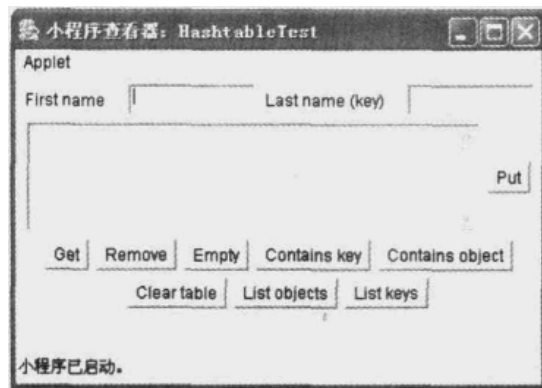


图 18.3 示例 Hashtable 类

下列语句:

```
object val = table.remove( lName.getText() );
```

使用 Hashtable 的方法 remove 从表中删除与关键字（作为参数给出）相关联的值。该方法返回一个 Object，它指向删除的值。如果没有与指定关键字相对应的值，则返回 null。

下列表达式:

```
table.isEmpty( )
```

如果 Hashtable 是空的，那么将返回 true；否则返回 false。

下列表达式:

```
table.containsKey( lName.getText() )
```

使用 Hashtable 的方法 containsKey 来判断作为参数给出的关键字是否在 Hashtable 中（即是否有与关键字相关的值）。如果存在，则该方法返回 true；否则返回 false。Hashtable 类也提供了 contains 方法，判断作为参数给出的 Object 是否在 Hashtable 中。下列语句:

```
table.clear();
```

清空 Hashtable。

下列表达式：

```
table.elements();
```

使用 Hashtable 的 elements 方法来获得 Hashtable 中值的一个 Enumeration。

下列表达式：

```
table.keys()
```

使用 Hashtable 的方法 keys 来获得 Hashtable 中关键字的一个 Enumeration。

如果要了解关于 Hashtable 方法的更多信息，请参看 Java API 的说明文档。

## 18.6 Date 类

管理日期和时间是一项大多数程序员难以意识到其复杂性的任务。Java 提供了 Date 类来帮助程序员标准地使用日期和时间，并且确保它们的完整性。我们在这里只给出一个简单的例子，建议读者对 Java API 进行更深入地探究。

Date 类包括大量的初始化 Date 对象的构造函数。Date 包括许多的方法，诸如 getYear、setYear、getMonth、setMonth、getDate、setDate、getDay、setDay、getHours、setHours、getMinutes、setMinutes、getSeconds、setSeconds 等。before 和 after 方法检查生成的 Date 对象是否在另一个作为参数给出的 Date 之前或之后；equals 方法将 Date 对象同另一个 Date 进行比较；hashCode 方法计算散列码，以便在 Hashtables 中插入和恢复 Date。

图 18.4 的程序给出了使用无参构造函数来构造一个 Date 对象的过程，它使用了系统日期和时间作为初始值；同时也给出了使用带有三个参数——年、月和日的构造函数来构造一个 Date 对象的过程。注意，代表一月的月份值是 0 而代表十二月的月份值是 11。程序中使用 toString 方法来把 Date 转换成 String 对象，以便显示。

```
1 // Fig. 18.4: DateTest.java
2 // Test class Date of the java.util package.
3 import java.util.Date;
4 import java.awt.Graphics;
5 import java.applet.Applet;
6
7 public class DateTest extends Applet {
8     Date system, user;
9
10    public void init()
11    {
12        system = new Date();
13        user = new Date( 96, 6, 4 );
14    }
15
16    public void paint( Graphics g )
17    {
18        g.drawString( "System date: " +
19                    system.toString(), 25, 25 );
20        g.drawString( "User defined date: " +
21                    user.toString(), 25, 40 );
```

```

22     }
23 }

```

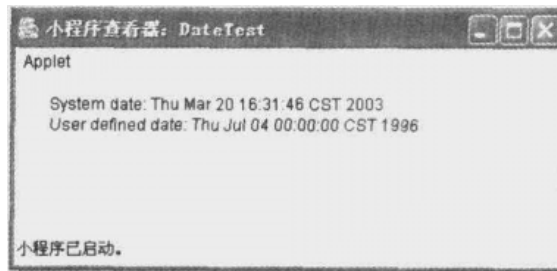


图 18.4 java.util 软件包中的 Date 类

## 18.7 Observable 类和 Observer 接口

在面向对象的系统中,经常重复出现一种常见的设计模板:一个对象由于某种原因改变了状态并且需要把有关的变化通知给其他一些对象,改变状态的对象应该是 Observable 类的一个子类。需要对其进行通知的对象称为观察者(observer),并且这些对象通过 Observer 接口进行操作。Observable 对象需要维护一个有关它的观察者的可编辑的表,如果 Observable 对象改变了状态,那么通过调用每个观察者的 update 方法来通知这些对象。

图 18.5 中的程序给出了一个 Observable 对象和一个 Observer 之间的交互作用。该程序由 applet 的 ObserverTest 类组成,该类扩展了 Applet 类并实现了 Observer 接口。ObserverTest 类定义了 Observer 接口的 update 方法,它将自动调用以便通知 applet, Clock 类的一个对象的状态改变了。Clock 类扩展了 Observable 类并且实现了 Runnable 接口,因此它能作为一个单独的线程执行。

```

1 // Fig. 18.5: ObserverTest.java
2 // Test of class Observable and interface Observer
3 // from the java.util package.
4 import java.util.*;
5 import java.awt.*;
6 import java.applet.Applet;
7
8 public class ObserverTest extends Applet implements Observer {
9     Clock c;
10    Label currentLabel;
11    TextField current;
12    Button alarmOn;
13    Thread clockThread;
14
15    public void init()
16    {
17        c = new Clock();
18        c.addObserver( this );
19        clockThread = new Thread( c );
20
21        currentLabel = new Label( "Current date and time: " );
22        current = new TextField( c.getDate(), 25 );
23        current.setEditable( false );
24        alarmOn = new Button( "Alarm on" );
25        add( currentLabel );

```

```
26         add( current );
27         add( alarmOn );
28
29         clockThread.start();
30     }
31
32     public boolean action( Event e, Object o )
33     {
34         showStatus( "Alarm set at " + c.setAlarm() );
35         return true;
36     }
37
38     public void update( Observable ob, Object arg )
39     {
40         current.setText( arg.toString() );
41         showStatus( "Alarm sounded, see new time" );
42     }
43 }
44
45 class Clock extends Observable implements Runnable {
46     Date d;
47     boolean alarmSet;
48
49     public Clock() { d = new Date(); }
50
51     public String setAlarm()
52     {
53         alarmSet = true;
54         d = new Date();
55         return d.toString();
56     }
57
58     public String getDate()
59     {
60         return d.toString();
61     }
62
63     public void run()
64     {
65         while ( true ) {
66             if ( alarmSet ) {
67                 try {
68                     Thread.sleep( 10000 );
69                 }
70                 catch( InterruptedException e ) {
71                     System.err.println( "Sleep interrupted" );
72                 }
73
74                 alarmSet = false;
75                 d = new Date(); // get new time
76                 setChanged();
77                 notifyObservers( d.toString() );
78             }
79         }
80     }
81 }
```

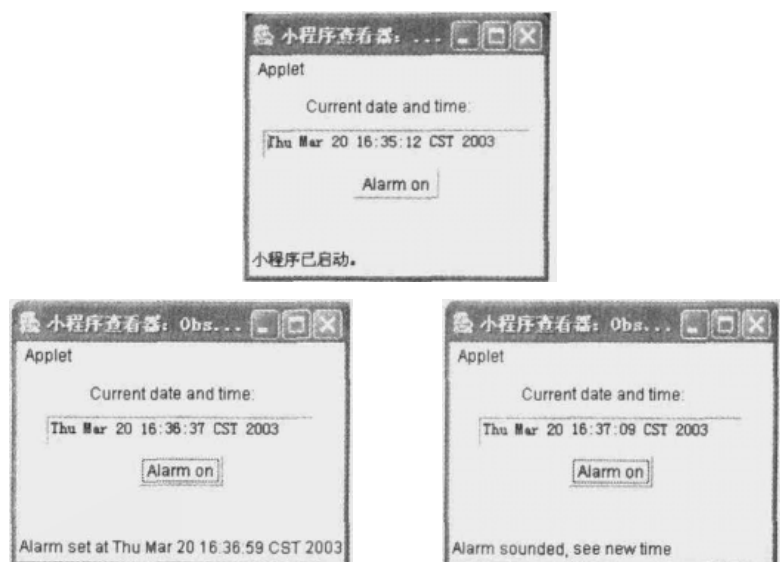


图 18.5 示例 Observable 类和 Observer 接口

当 applet 执行时，就会显示带有一个文本字段和一个按钮的图形用户界面。程序中的 `init` 方法实例化一个 `Clock` 对象 `c`。下列语句：

```
c.addObserver( this );
```

把进程加入到 `Clock` 对象 `c` 的 `Observers` 表中。

接着实例化线程 `clockThread` 并用 `Clock` 对象 `c` 来初始化，这样 `Clock` 就能作为一个单独的线程来执行。当实例化文本字段时，将由 `c.getDate()` 的值对其进行初始化，如果此时已经创建了 `Clock` 对象，那么该函数就返回一个包含日期和时间的 `String`。如果用户按下按钮，就会激活 `Clock` 对象 `c` 的 `setAlarm` 方法。因此把 `Clock` 对象的 `alarmSet` 实例变量置为 `true`，以便指示在 10 秒之内就会响铃。当 `Clock` 对象从睡眠中苏醒过来之后，就通过创建一个新的 `Date` 对象来捕获当前的日期和时间，然后将激活 `Observable` 类的 `setChanged` 方法，以指示 `Clock` 对象的状态已经改变了。接着，激活 `Observable` 的 `notifyObservers` 方法，并且将其传递给一个代表新的日期和时间的 `String`。该方法自动调用所有观察者的 `update` 方法（即本程序中的 `applet`）。

`Observer` 的方法 `update` 有两个参数——一个指向其状态信息已改变的 `Observable` 对象，另一个指向 `Object`，这个 `Object` 将传递给 `Observable` 对象的 `notifyObservers` 方法。`Object` 的类型一般依赖于所在的程序。在本程序中，它是表示新的日期和时间的 `String`。这个例子中的 `update` 方法在文本字段中显示新的日期和时间，并在状态栏区域显示一条信息以指出闹铃已经响过了。

`Observable` 类还提供一些没有在图 18.5 中使用的方法，下面简要概述使用 `Clock` 对象 `c` 作为 `Observable` 对象的其他方法。下列语句：

```
c.notifyObservers()
```

通知 `c` 的 `Observers` 表中的所有观察者，`c` 的状态已经改变，这将调用每个 `Observer` 对象的 `update` 方法。在这种情况下，`update` 方法的第二个参数为 `null`。下列语句：

```
c.deleteObservers()
```

将删除 `c` 的 `Observers` 表中的观察者。下列语句：

```
c.countObservers()
```

将返回 `c` 的 `Observers` 表中的观察者的数量。

下列语句：

```
c.clearChanged()
```

对指示已改变的 `Observable` 对象的状态标志进行复位，该方法将由 `notifyObservers` 方法自动调用。当已经设置 `c` 的状态标志时，下列语句：

```
c.hasChanged()
```

返回 `true`，否则返回 `false`。

## 18.8 Properties 类

一个 `Properties` 对象是一个永久的 `Hashtable` 对象。所谓永久，是指能将 `Hashtable` 对象写到一个输出流并保存在一个文件中，然后可以把它读回一个输入流。`Properties` 类扩展了 `Hashtable` 类，因此，我们讨论的图 18.3 中的方法也能由 `Properties` 对象使用。`Properties` 对象的关键字和值必须是 `String` 类型。`Properties` 类提供了其他一些方法，它们出现在图 18.6 的应用程序中。

下列语句：

```
table = new Properties();
```

使用无参构造函数来创建一个空的没有默认属性的 `Properties` 表，带有默认属性值的 `Properties` 对象的一个引用也可以传递给该构造函数，这个版本对于维护与下面介绍的 `getProperty` 方法一起使用的一系列默认值是很有用的。

下列语句：

```
object val = table.getProperty( propName.getText() );
```

使用 `Properties` 方法 `getProperty` 来定位作为参数给出的关键字的值。如果在 `Properties` 对象中找不到该关键字，就使用默认的 `Properties` 对象（如果有一个）。递归执行该过程，直到不再有默认的 `Properties` 对象（记住，每个 `Properties` 对象都能由默认的 `Properties` 对象初始化），即返回指针 `null`。所提供方法的第二个版本带有两个参数，其中第二个参数是当 `getProperty` 方法不能定位关键字时所返回的默认值。

下列语句：

```
table.save( output , "Sample Properties" );
```

使用 `Properties` 方法 `save` 把 `Properties` 对象的内容存放到作为第一个参数给出的 `OutputStream` 对象（这里是一个 `FileOutputStream`）中。`Properties` 类也提供了使用 `PrintStream` 作为参数的方法，该方法对于显示属性集合是很有用的。

---

```
1 // Fig. 18.6: PropertiesTest.java
2 // Demonstrates class Properties of the java.util package.
3 import java.io.*;
4 import java.util.*;
5 import java.awt.*;
6
7 public class PropertiesTest extends Frame {
8     Properties table;
```

```
9      FileInputStream input;
10     FileOutputStream output;
11
12     Label prompt1, prompt2;
13     TextField propName, propVal, result;
14     TextArea display;
15     Button put, clear, getProperty, save, load;
16
17     public PropertiesTest()
18     {
19         super( "Properties Test" );
20         table = new Properties();
21
22         prompt1 = new Label( "Property name (key)" );
23         propName = new TextField( 10 );
24         prompt2 = new Label( "Property value" );
25         propVal = new TextField( 10 );
26         display = new TextArea( 4, 35 );
27         put = new Button( "Put" );
28         clear = new Button( "Clear" );
29         getProperty = new Button( "Get property" );
30         save = new Button( "Save" );
31         load = new Button( "Load" );
32         result = new TextField( 35 );
33         result.setEditable( false );
34         setLayout( new FlowLayout() );
35         add( prompt1 );
36         add( propName );
37         add( prompt2 );
38         add( propVal );
39         add( display );
40         add( put );
41         add( clear );
42         add( getProperty );
43         add( save );
44         add( load );
45         add( result );
46         resize( 330, 225 );
47         show();
48     }
49
50     public boolean handleEvent( Event e )
51     {
52         if ( e.id == Event.WINDOW_DESTROY ) {
53             hide();
54             dispose();
55             System.exit( 0 );
56         }
57
58         return super.handleEvent( e );
59     }
60
61     public boolean action( Event e, Object o )
62     {
63         if ( e.target == put ) {
64             Object val = table.put( propName.getText(),
65                                     propVal.getText() );
66
67             if ( val == null )
68                 showStatus( "Put: " + propName.getText() + " " +
69                             propVal.getText() );
```

```

70         else
71             showStatus( "Put: " + propName.getText() + " " +
72                         propName.getText() + "; Replaced: " +
73                         val.toString() );
74     }
75     else if ( e.target == clear ) {
76         table.clear();
77         showStatus( "Table in memory cleared" );
78     }
79     else if ( e.target == getProperty ) {
80         Object val = table.getProperty( propName.getText() );
81
82         if ( val != null )
83             showStatus( "Get property: " + propName.getText() +
84                         " " + val.toString() );
85         else
86             showStatus( "Get: " + propName.getText() +
87                         " not in table" );
88     }
89     else if ( e.target == save ) {
90         try {
91             output = new FileOutputStream( "props.dat" );
92             table.save( output, "Sample Properties" );
93             output.close();
94         }
95         catch( IOException ex ) {
96             showStatus( ex.toString() );
97         }
98     }
99     else if ( e.target == load ) {
100         try {
101             input = new FileInputStream( "props.dat" );
102             table.load( input );
103             input.close();
104         }
105         catch( IOException ex ) {
106             showStatus( ex.toString() );
107         }
108     }
109
110     listProperties();
111     return true;
112 }
113
114 public void listProperties()
115 {
116     StringBuffer buf = new StringBuffer();
117     String pName, pVal;
118
119     for ( Enumeration enum = table.propertyNames();
120           enum.hasMoreElements(); ) {
121         pName = enum.nextElement().toString();
122         pVal = table.getProperty( pName );
123         buf.append( pName ).append( '\t' );
124         buf.append( pVal ).append( '\n' );
125     }
126
127     display.setText( buf.toString() );
128 }
129
130 public void showStatus( String s )

```



```
131    {  
132        result.setText( s );  
133    }  
134  
135    public static void main( String args[ ] )  
136    {  
137        PropertiesTest p = new PropertiesTest();  
138    }  
139 }
```

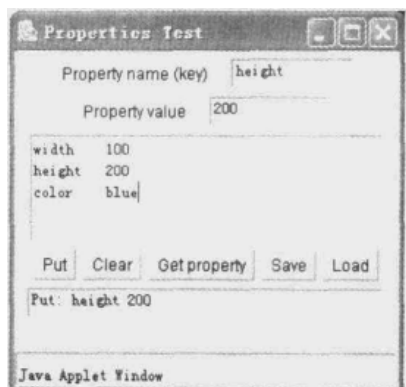


图 18.6 示例 Properties 类

#### 测试与调试提示 18.2

为了调试的目的，使用 Properties 的方法 list 来显示 Properties 对象的内容。

下列语句：

```
table.load( input );
```

使用了 Properties 的 load 方法，从作为第一个参数给出的 InputStream 中恢复 Properties 对象的内容（这里是 FileInputStream）。

下列表达式：

```
table.propertyName( )
```

使用了 Properties 的 propertyName 方法，获得属性名的一个 Enumeration，每个属性值都能通过 getProperty 方法进行描述。

## 18.9 Random 类

我们在第 4 章中讨论过随机数的生成，那时我们使用的是 Math 类的 random 方法。在 Random 类中，Java 提供了其他扩展的随机数生成方法，这里我们简要地介绍一下 API 调用。

可以使用：

```
Random r = new Random( );
```

创建一个新的随机数产生器。这种形式采用时间作为随机数产生器的“种子”（seed），因此每次调用时的种子都不同，这样每次都能产生不同的随机数序列。

为了创建一个具有重复性特征的伪随机数产生器，我们采用下面的语句：

```
Random r = new Random( seedValue );
```

这种形式每次都使用相同的 longseedValue，随后就会产生相同的随机数序列。

#### 测试与调试提示 18.3

如果使用相同的种子值就会出现重复的随机数，这对于测试和调试是很有用的。当一个程序还处在开发阶段时，最好使用 Random( seedValue ) 形式来生成重复的随机数序列。如果出现错误，就进行改正并用同样的 seedValue 进行测试，这样就可以重新构造同引起错误时的序列完全一样的随机数序列。全部改正完错误后，就在每次程序运行时使用 Random() 形式来产生新的随机数序列。

下列调用：

```
r.setSeed( seedValue );
```

在任何时候重新设定 r 的种子值。

下列调用：

```
r.nextInt( )  
r.nextLong( )
```

产生一致分布的随机整数。可以使用 Math.abs 来得到用 nextInt 产生的数的绝对值，这样就能给出一个从 0 到接近 200 万的范围内的数值，然后使用 % 运算符来对该数值进行取模操作。例如，滚动一个有 6 面且各面值为 1~6 的骰子，如果用 6 进行取模，就能得到一个 0 到 5 范围内的整数，然后加 1 来改变这个值就能产生 1 到 6 范围之间的数值。表达式如下所示：

```
Math.abs( r.nextInt() ) % 6 + 1
```

下列调用：

```
r.nextFloat( )  
r.nextDouble( )
```

产生  $0.0 \leq x < 1.0$  范围内的一致分布的值。

下列调用：

```
r.nextGaussian( )
```

采用 Gaussian 分布（期望值是 0.0，方差是 1.0）的概率密度来产生一个双精度浮点数。

## 18.10 位处理和位运算符

Java 为需要考虑称为“位和字节”操作的程序员提供扩展的位处理功能。操作系统、测试配置的软件、网络软件以及其他许多软件都要求程序员直接和硬件进行通信。在这一节和下一节中，我们将讨论 Java 的位处理功能，其中介绍了许多 Java 的位运算符并示例它们的使用。

所有数据在计算机内部都被表示成位的序列，每一位都具有值 0 或值 1。在许多系统中，一个 8 位的序列组成一个字节——类型为 byte 的变量的标准存储单元。其他数据类型将占用更多的字节，位运算符用来处理整数操作数（byte、char、short、int 和 long）的位。

注意，这一节对位运算符的讨论涉及到整数操作数的二进制表示，关于二进制（基数为 2）数值系统的详细解释请参见附录 C “数值系统”。

位运算符包括：位与（&）、位或（|）、位异或（^）、左移（<<）、带符号扩展右移（>>）、带零扩展右移（>>>）和位取反（~）。位与、位或和位异或运算符按位比较它们的两个操作数。如果两个操作数中相应的位都是 1，那么位与运算符就将结果中的该位置为 1。如果两个操作数相应的位中有一个（或两个）是 1，那么位或运算符就把结果中的该位置为 1。如果两个操作数相应的位中只有一个是 1，那么位异或运算符就把结果中的该位置为 1。左移运算符把它左操作数的位序列向左移动由右操作数指定的位数。带符号扩展右移运算符把它左操作数的位序列向右移动由右操作数指定的位数——如果左操作数是负的，那么就把 1 序列从左边移入左操作数；否则，就将 0 序列从左边移入。带零扩展右移运算符把它左操作数的位序列向右移动由右操作数指定的位数——0 序列从左边移入左操作数。位取反运算符把操作数中的所有 0 变为结果中的 1，所有 1 变为结果中的 0。下面的例子中给出了对每个位运算符的详细讨论，图 18.7 中总结了位运算符。

运算符	名称	描述
&	位与	如果两个操作数中相应的位都是 1，那么就把结果中该位置为 1
	位或	如果两个操作数中相应的位至少有一位是 1，那么就把结果中该位置为 1
^	位异或	如果两个操作数中相应的位仅有一位是 1，那么就把结果中该位置为 1
<<	左移	把第一个操作数的位序列向左移动由第二个操作数指定的位数，用 0 序列填充右边
>>	带符号扩展右移	把第一个操作数的位序列向右移动由第二个操作数指定的位数。如果第一个操作数是负的，就从左边移入 1 序列；否则从左边移入 0 序列
>>>	带零扩展右移	把第一个操作数的位序列向右移动由第二个操作数指定的位数；从左边移入 0 序列
~	位取反	所有的 0 变为 1，所有的 1 变为 0

图 18.7 位运算符

使用位运算符时，显示所得结果的二进制表示，以便说明这些运算是很有用的。图 18.8 中的程序允许用户在一个文本字段中输入一个整数并按下回车键，从而显示不同的结果。

```

1 // Fig. 18.8: PrintBits.java
2 // Printing an unsigned integer in bits
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class PrintBits extends Applet {
7     Label prompt, result;
8     TextField input, output;
9
10    public void init()
11    {
12        prompt = new Label( "Enter an integer " );
13        input = new TextField( 10 );
14        result = new Label( "The integer in bits is" );
15        output = new TextField( 32 );
16        output.setEditable( false );
17        add( prompt );
18        add( input );
19        add( result );
20        add( output );
21    }
22
23    public boolean action( Event e, Object o )
24    {
25        if ( e.target == input ) {

```

```

26         int val = Integer.parseInt( o.toString() );
27         output.setText( getBits( val ) );
28     }
29
30     return true;
31 }
32
33 public String getBits( int value )
34 {
35     int displayMask = 1 << 31;
36     StringBuffer buf = new StringBuffer( 35 );
37
38     for ( int c = 1; c <= 32; c++ ) {
39         buf.append(
40             ( value & displayMask ) == 0 ? '0' : '1' );
41         value <<= 1;
42
43         if ( c % 8 == 0 )
44             buf.append( ' ' );
45     }
46
47     return buf.toString();
48 }
49 }

```



图 18.8 显示用位序列表示的一个整数

程序中的 `action` 方法从文本字段中读入 `String`, 把它转换为一个整数并激活 `getBits` 方法 (第 33 行), 获得该整数用 `String` 表示的位序列, 得到的结果显示在输出文本字段中。整数用二进制表示, 每 8 位为一组。 `displayBits` 方法使用了位与运算符, 从而把变量 `value` 和 `displayMask` 组合起来。通常, 位与运算符和一个称为“掩码” (某些特定位设为 1 的整数值) 的操作数一起使用。在搜索值中的其他位时, 掩码用来隐藏某些位。在 `getBits` 方法中, 掩码变量 `displayMask` 将赋值为 `1 << 31` (10000000 00000000 00000000 00000000)。左移运算符把 1 从 `displayMask` 中的低位 (最右边) 移向高位 (最左边), 并用 0 序列从右边填充。下列语句:

```
buf.append( ( value & displayMask ) == 0 ? '0' : '1' );
```

说明把变量 `value` 中当前最左边的位 (0 或者 1) 添加到 `StringBuffer` 类的 `buf` 中。假设 `value` 是 4000000000 (11101110 01101011 00101000 00000000)。当 `value` 和 `displayMask` 通过 “&” 进行组合之后, 变量 `value` 中除了最高位之外所有的位都被“掩盖”(隐藏), 这是因为任何位使用 0 进行 “&” 操作后都变成了 0。如果最左边的位是 1, 那么 `value & displayMask` 就为值 1, 并且添加 1 (否则添加 0)。然后, 表达式 “`value << 1`” 将变量 `value` 左移一位 (这与 `value = value << 1` 等价)。变量 `value` 中的每一位都重复这些操作。在 `getBits` 方法的最后, 通过下列语句:

```
buf.toString();
```

把 `StringBuffer` 转换为 `String`, 然后从方法返回该 `String`。图 18.9 总结了通过位与运算符 (&) 组合两位得到的结果。

#### 常见编程错误 18.1

把逻辑与运算符 (&&) 误用为位与运算符 (&).

位 1	位 2	位 1 & 位 2
0	0	0
1	0	0
0	1	0
1	1	1

图 18.9 使用位与运算符 (&) 组合两位得到的结果

图 18.10 中的程序描述了位与运算符、位或运算符、位异或运算符以及位取反运算符的使用。该程序使用 `getBits` 方法来得到一个整数值的 `String` 表示。程序允许用户在文本字段中输入值并按下回车键 (因为是二元运算符, 所以必须输入两个值)。用户可以按下想要进行的操作, 然后就将显示整数表示和位序列表示的结果。

```

1 // Fig. 18.10: MiscBitOps.java
2 // Using the bitwise AND, bitwise inclusive OR, bitwise
3 // exclusive OR, and bitwise complement operators.
4 import java.awt.*;
5 import java.applet.Applet;
6
7 public class MiscBitOps extends Applet {
8     Label prompt1, prompt2, in1, in2, res, bitsLabel;
9     TextField input1, input2, result, bits1, bits2, bits3;
10    Button and, inclusiveOr, exclusiveOr, complement;
11    Panel inputPanel, bitsPanel, buttonPanel;
12
13    public void init()
14    {
15        setLayout( new BorderLayout() );
16        inputPanel = new Panel();
17        inputPanel.setLayout( new GridLayout( 4, 2 ) );
18        prompt1 = new Label( "Enter 2 ints" );
19        prompt2 = new Label( "" );
20        in1 = new Label( "Value 1" );
21        input1 = new TextField( 8 );
22        in2 = new Label( "Value 2" );
23        input2 = new TextField( 8 );
24        res = new Label( "Result" );
25        result = new TextField( 8 );

```

```

26      result.setEditable( false );
27      inputPanel.add( prompt1 );
28      inputPanel.add( prompt2 );
29      inputPanel.add( in1 );
30      inputPanel.add( input1 );
31      inputPanel.add( in2 );
32      inputPanel.add( input2 );
33      inputPanel.add( res );
34      inputPanel.add( result );
35      bitsPanel = new Panel();
36      bitsPanel.setLayout( new GridLayout( 4, 1 ) );
37      bitsLabel = new Label( "Bit representations" );
38      bits1 = new TextField( 30 );
39      bits1.setEditable( false );
40      bits2 = new TextField( 30 );
41      bits2.setEditable( false );
42      bits3 = new TextField( 30 );
43      bits3.setEditable( false );
44      bitsPanel.add( bitsLabel );
45      bitsPanel.add( bits1 );
46      bitsPanel.add( bits2 );
47      bitsPanel.add( bits3 );
48      buttonPanel = new Panel();
49      and = new Button( "AND" );
50      inclusiveOr = new Button( "Inclusive OR" );
51      exclusiveOr = new Button( "Exclusive OR" );
52      complement = new Button( "Complement" );
53      buttonPanel.add( and );
54      buttonPanel.add( inclusiveOr );
55      buttonPanel.add( exclusiveOr );
56      buttonPanel.add( complement );
57      add( "West", inputPanel );
58      add( "East", bitsPanel );
59      add( "South", buttonPanel );
60  }
61
62  public boolean action( Event e, Object o )
63  {
64      if ( e.target == complement ) {
65          input2.setText( "" );
66          bits2.setText( "" );
67          int val = Integer.parseInt( input1.getText() );
68          result.setText( Integer.toString( val ) );
69          bits1.setText( getBits( val ) );
70          bits3.setText( getBits( val ) );
71      }
72      else {
73          int val1 = Integer.parseInt( input1.getText() );
74          int val2 = Integer.parseInt( input2.getText() );
75
76          bits1.setText( getBits( val1 ) );
77          bits2.setText( getBits( val2 ) );
78
79          if ( e.target == and ) {
80              result.setText( Integer.toString( val1 & val2 ) );
81              bits3.setText( getBits( val1 & val2 ) );
82          }
83          else if ( e.target == inclusiveOr ) {
84              result.setText( Integer.toString( val1 | val2 ) );
85              bits3.setText( getBits( val1 | val2 ) );
86          }

```

```

87         else if ( e.target == exclusiveOr ) {
88             result.setText( Integer.toString( val1 ^ val2 ) );
89             bits3.setText( getBits( val1 ^ val2 ) );
90         }
91     }
92
93     return true;
94 }
95
96 public String getBits( int value )
97 {
98     int displayMask = 1 << 31;
99     StringBuffer buf = new StringBuffer( 35 );
100
101     for ( int c = 1; c <= 32; c++ ) {
102         buf.append( ( value & displayMask ) == 0 ? '0' : '1' );
103         value <<= 1;
104
105         if ( c % 8 == 0 )
106             buf.append( ' ' );
107     }
108
109     return buf.toString();
110 }
111 }

```



图 18.10 示例位与运算符、位或运算符、位异或运算符以及位取反运算符

图 18.10 中的输出窗口显示了值 65535 和值 1 用位与运算符 (&) 组合后得到的结果。值 65535 中除了最低位外所有的位都通过与值 1 的 “&” 操作而被“掩盖”(隐藏)。

位或运算符用于把一个操作数中的某些指定的位置为 1。图 18.10 中的第二个输出窗口显示了值 15 和值 241 用位或运算符组合后得到的结果——结果是 255。图 18.11 总结了两个位通过位或运算符组合后得到的结果。

位 1	位 2	位 1   位 2
0	0	0
1	0	1
0	1	1
1	1	1

图 18.11 用位或运算符 (|) 组合两个位后的结果

**常见编程错误 18.2**

把逻辑或运算符 (||) 误用为位或运算符 (|)

如果两个操作数中只有一个操作数的某一位是 1, 那么位异或运算符就把结果中相应的位置为 1。图 18.10 中的第三个输出窗口显示了值 139 和 199 通过位异或运算符组合后得到的结果——该结果是 76。图 18.12 总结了两个位通过位异或运算符组合后得到的结果。

位 1	位 2	位 1 ^ 位 2
0	0	0
1	0	1
0	1	1
1	1	0

图 18.12 使用位异或运算符 (^) 组合两个位后的结果

位取反运算符 (~) 把操作数中的所有 0 变为结果中的 1, 所有 1 变为结果中的 0——也可称为“求出该值的反码”。图 18.10 中的第四个窗口显示了对值 21845 求反码后得到的结果。结果为 -21846。

图 18.13 中的程序描述了左移运算符 (<<)、带符号扩展右移运算符 (>>) 和带零扩展右移运算符 (>>>)。getBits 方法用于获得整数值的 String 表示。该程序允许用户在文本字段中输入一个整数并按下回车键, 整数的位表示显示在另一个文本字段中。

为每个移位运算符都提供一个按钮。当用户按下每个按钮时, 该整数的位序列就向左或向右移动一位。新的整数值和新的位序列则显示在文本字段中。

```

1 // Fig. 18.13: BitShift.java
2 // Using the bitwise shift operators.
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class BitShift extends Applet {
7     Label prompt;
8     TextField value, bits;
9     Button left, rightSign, rightZero;
10
11     public void init()
12     {
13         prompt = new Label( "Integer to shift " );
14         value = new TextField( 8 );
15         bits = new TextField( 30 );
16         bits.setEditable( false );
17         left = new Button( "<<" );
18         rightSign = new Button( ">>" );
19         rightZero = new Button( ">>>" );
20         add( prompt );
21         add( value );

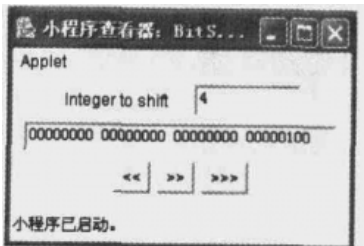
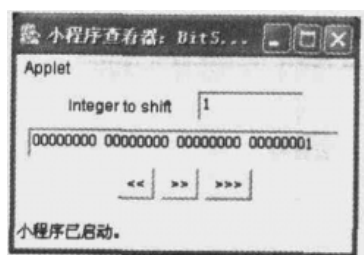
```



```

22         add( bits );
23         add( left );
24         add( rightSign );
25         add( rightZero );
26     }
27
28     public boolean action( Event e, Object o )
29     {
30         int val = Integer.parseInt( value.getText() );
31
32         if ( e.target instanceof Button ) {
33             if ( e.target == left )
34                 val <<= 1;
35             else if ( e.target == rightSign )
36                 val >>= 1;
37             else if ( e.target == rightZero )
38                 val >>>= 1;
39
40             value.setText( Integer.toString( val ) );
41         }
42
43         bits.setText( getBits( val ) );
44         return true;
45     }
46
47     public String getBits( int value )
48     {
49         int displayMask = 1 << 31;
50         StringBuffer buf = new StringBuffer( 35 );
51
52         for ( int c = 1; c <= 32; c++ ) {
53             buf.append( ( value & displayMask ) == 0 ? '0' : '1' );
54             value <<= 1;
55
56             if ( c % 8 == 0 )
57                 buf.append( ' ' );
58         }
59
60         return buf.toString();
61     }
62 }

```



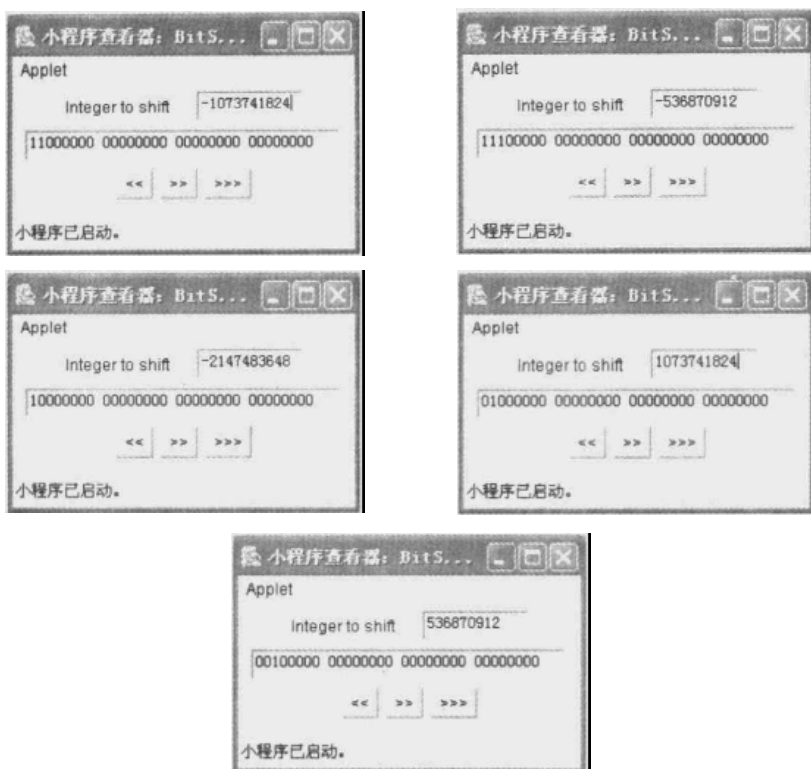


图 18.13 示例位移运算符

带符号扩展右移运算符(>>)把它的左操作数的位序列向右移动由它的右操作数指定的位数。完成一次右移会导致左边有空缺的位,如果左操作数是正的,空缺位就补0,如果是负的,就补1,所有从右边移走的1都丢失了。第五个和第六个输出窗口显示了把第四个输出窗口中的值右移(带符号扩展)两次所得到的结果。

带零扩展右移运算符(>>>)把它的左操作数的位序列向右移动由它的右操作数指定的位数。完成一次右移后所引起的左边的空缺位用0填充,所有从右边移走的1都丢失了。第八个和第九个输出窗口显示了把第七个输出窗口中的值右移(带零扩展)两次所得到的结果。

每个位运算符(位补运算符除外)都有一个相应的赋值运算符。图 18.14 给出了这些位赋值运算符,它们的用法同第 2 章介绍的算术赋值运算符的用法类似。

位赋值运算符	
&=	位与赋值运算符
=	位或赋值运算符
^=	位异或赋值运算符
<<=	左移赋值运算符
>>=	带符号扩展右移赋值运算符
>>>=	带零扩展右移赋值运算符

图 18.14 位赋值运算符

## 18.11 BitSet 类

BitSet类使创建和使用位集合变得更加容易。位集合对于表示一组布尔标志是很有用的。BitSets是动态的。如果需要增加更多的位,那么 BitSet 对象将会扩充,以包括这些附加的位。

下列表达式:

```
Bit Setb = new BitSet ( );
```

可以创建一个初始值为空的 BitSet 对象。

下列表达式:

```
Bit Setb = new BitSet (size);
```

可以创建一个有 size 位的 BitSet 对象。下列表达式:

```
b.set ( bitNumber ) ;
```

可以把 BitSet 中的位 bitNumber 置为 “on” 下列表达式:

```
b.clear ( bitNumber );
```

可以把 BitSet 中的位 bitNumber 置为 “off”。使用下列表达式:

```
b.get( bitNumber ) ;
```

将得到 BitSet b 中的位 bitNumber。如果该位为 on 则返回 true; 如果该位为 off 则返回 false。

下列表达式:

```
b.and ( b1 );
```

将在 BitSet b 和 b1 之间使用一次按位的逻辑与操作, 结果放在 b 中。通过下列语句:

```
b.or ( b1 );  
b.xor( b2 );
```

可以完成位逻辑或和位逻辑异或。下列表达式:

```
b.size( )
```

将返回 BitSet b 的大小。下列表达式:

```
b.equals( b1 )
```

将比较两个 BitSet 对象是否相等。下列表达式:

```
b.clone( )
```

将复制一个 BitSet 并返回一个指向新的 BitSet 的 Object。下列表达式:

```
b.toString( )
```

可以把 BitSet b 转换为一个 String, 这有助于程序调试。下列表达式:

```
b.hashCode( )
```

将提供一个用于在散列表中存储和检索 BitSet 对象的有用的散列码。

图 18.15 的程序再次回到女神的筛子这个问题, 以找出我们在练习 5.27 中讨论过的素数。这里使用一个 BitSet 代替数组来完成算法。程序在一个文本区域中显示出从 1 到 1 023 之间所有的素数; 并且提供一个文本字段, 用户可以在其中输入 1 到 1 023 之间的任意一个数以判断该数是否为一个素数 (在这种情况下, 信息会显示在 applet 中的状态区域内)。

```

1 // Fig. 18.15: BitSetTest.java
2 // Using a BitSet to demonstrate the Sieve of Eratosthenes.
3 import java.applet.Applet;
4 import java.awt.*;
5 import java.util.*;
6
7 public class BitSetTest extends Applet {
8     BitSet sieve;
9     Panel inputPanel;
10    Label inputLabel;
11    TextField input;
12    TextArea primes;
13
14    public void init()
15    {
16        sieve = new BitSet( 1024 );
17
18        setLayout( new BorderLayout() );
19        inputLabel = new Label( "Enter a value from 1 to 1023" );
20        input = new TextField( 10 );
21        inputPanel = new Panel();
22        inputPanel.add( inputLabel );
23        inputPanel.add( input );
24        add( "North", inputPanel );
25
26        primes = new TextArea();
27        add( "Center", primes );
28
29        // set all bits from 1 to 1023
30        int size = sieve.size();
31
32        for ( int i = 1; i < size; i++ )
33            sieve.set( i );
34
35        // perform Sieve of Eratosthenes
36        int finalBit = (int) Math.sqrt( sieve.size() );
37
38        for ( int i = 2; i < finalBit; i++ )
39            if ( sieve.get( i ) )
40                for ( int j = 2 * i; j < size; j += i )
41                    sieve.clear( j );
42
43        int counter = 0;
44
45        for ( int i = 1; i < size; i++ )
46            if ( sieve.get( i ) ) {
47                primes.appendText( String.valueOf( i ) );
48                primes.appendText(
49                    ++counter % 7 == 0 ? "\n" : "\t" );
50            }
51    }
52
53    public boolean action( Event e, Object o )
54    {
55        int val = Integer.parseInt( input.getText() );
56
57        if ( sieve.get( val ) )
58            showStatus( val + " is a prime number" );
59        else
60            showStatus( val + " is not a prime number" );
61    }
62

```

```

62
63         return true;
64     }
65 }

```

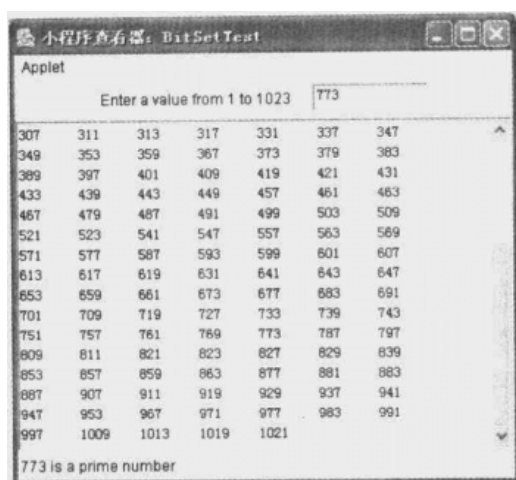


图 18.15 使用 BitSet 的“女神的筛子”程序

下列语句:

```
sieve = new Bit Set( 1024 );
```

创建了一个有 1 024 位的 BitSet 对象, 在本程序中我们不考虑处于位置 0 的位。在 init 方法后把 BitSet 中的所有位都置为 on, 下列代码:

```

// perform Sieve of Eratosthenes
int finalBit = (int) Math.sqrt( sieve.size() );

for ( int i = 2 ; i < finalBit; i++ )
    if ( sieve.get( i ) )
        for (int j = 2 * i ; j < size ; j += i )
            sieve.clear( j );

```

给出了 1 到 1 023 之间所有的素数。整数 finalBit 用于判断算法是否完成。基本算法如下: 如果一个数除了 1 和自己之外, 再不能被其他数整除, 那么它就是素数。1 是素数。从 2 开始, 一旦我们知道一个数是素数, 就能排除该数所有的倍数。这样, 我们可以排除 4、6、8 等。3 可以被 1 和自己整除, 因此我们可以排除 3 的所有倍数 (记住, 所有的偶数已经排除)。

## 小结

- Vector 类提供了可以动态改变大小的数组。
- 在任何时候, Vector 都包括少于或等于它的容量的一定数量的元素, 这个容量是为数组预先保留的。
- 如果一个 Vector 需要增大, 它可以按照系统默认或用户的要求增大。
- 如果没有注明数组需要扩大的容量, 系统自动将 Vector 现有容量大小增加一倍。
- Vector 设计用来存放 Object 的引用。如果要在 Vector 中存储原始数据类型的值, 就必须使用相应类型的包装类 (Integer、Long、Float、Double、Boolean 和 Character) 来创建包含原始数

据类型值的对象。

- Vector类提供了一种构造函数。不带参数的构造函数创建一个空的Vector。带有一个参数的构造函数把参数指定的值作为初始容量来创建一个Vector。带有两个参数的构造函数把第一个参数指定的值作为初始容量来创建一个Vector，第二个参数指定容量的增值。
- Vector的addElement方法可将参数添加到Vector的尾部，insertElementAt方法可将一个元素插入到指定位置，setElementAt方法则可将元素放在指定的位置上。
- Vector的removeElement方法将删除与其参数匹配的的第一个元素，removeAllElement方法可将Vector中的所有元素删除，removeElementAt方法则将删除指定位置上的元素。
- Vector的firstElement方法返回第一个元素的引用，lastElement方法则返回最后一个元素的引用。
- Vector的isEmpty方法可判断Vector是否为空，contains方法可判断Vector是否包含作为参数的指定的searchKey元素，indexOf方法则取得其参数的第一个位置的索引。如果Vector中找不到该参数，则该方法返回-1。
- Vector的trimToSize方法可用于调整Vector的容量，使其等于Vector的大小。size方法和capacity方法分别判断当前Vector中元素的个数，以及在不需分配更多空间的前提下能够存储在Vector中的元素个数。
- Vector的elements方法可返回包含Vector中元素的一个Enumeration的引用。
- Enumeration的hasMoreElements方法可用于判断是否有更多的元素。nextElement方法可用于返回下一个元素的引用。
- Stack类是对Vector类的扩展。Stack的push方法把它的参数加到堆栈的顶端，pop方法移出栈顶元素，peek方法在不移走元素的前提下返回指向堆栈顶端元素的一个Object，empty方法判断堆栈是否是为空。
- Dictionary将关键字转换成相应的值。
- 存储和检索信息时，如果想把关键字转换成惟一的数组下标，那么选择散列技术就意味着选择了一种高速方案。负荷因子是散列表中已占用的单元数目与散列表大小的比值，这个比值越接近1.0，系统冲突产生的机会就越大。
- 不带参数的Hashtable的构造函数创建一个Hashtable，它的默认容量是101，默认负荷因子是0.75。带一个参数的Hashtable构造函数指定初始的容量，带两个参数的构造函数分别指定初始容量和负荷因子。
- Hashtable的put方法可向Hashtable中增加一个关键字和一个值。get方法可定位与指定的关键字相关联的值，remove方法可从表中删除与指定的关键字相关联的值，isEmpty方法可判断表是否为空。
- Hashtable的containsKey方法可用于判断作为参数给出的关键字是否在Hashtable中（即是否有与关键字相关联的值）。contains方法可用于判断作为参数给出的Object是否在Hashtable中。clear方法用于清空Hashtable。elements方法用于获得值的一个Enumeration。keys方法用于获得关键字的一个Enumeration。
- Date类包括大量的对日期进行初始化的构造函数，其中包含以下方法：getYear，setYear，getMonth，setMonth，getDate，setDate，getDay，setDay，getHours，setHours，getMinutes，setMinutes，getSeconds，setSeconds，等等。before方法和after方法可检查生成的Date对象是否在作为参数给出的另一个Date之前或之后，toString方法可把Date对象转换成String。
- 面向对象的系统中经常重复出现一种设计模板：一个对象改变了状态并且需要把有关改变通知给其他一些对象，改变状态的对象应该是Observable类的一个子类，需要通知的对象

在 Observer 接口上进行操作

- Observable 类的 addObserver 方法用于把一个 Observer 加入到一个放置 Observer 的 Observable 对象表中。setChanged 方法指示 Observable 对象的状态已经改变了。
- Observable 类的 notifyObservers 方法可以自动为其所有的观察者激活 update 方法。update 方法带有两个参数——一个指向其状态信息已改变了的 Observable 对象；另一个指向 Object，这个 Object 将传递给 Observable 对象的 notifyObservers 方法，deleteObservers 方法可用于删除一个 Observable 对象的观察者。
- Observable 类的 clearChanged 方法可用于对指明已改变的 Observable 对象的状态标志进行复位，该方法由 notifyObservers 方法自动调用。hasChanged 方法用于判断一个 Observable 对象是否已经改变。countObservers 方法用于返回观察者的数量。
- Properties 对象是永久的 Hashtable 对象。Properties 类扩展了 Hashtable 类。Properties 对象的关键字和值必须是 String 类型。
- Properties 的无参数构造函数创建一个空的没有默认属性的 Properties 表。也可以将带有默认属性值的 Properties 对象的引用传递给构造函数。
- Properties 类的 getProperty 方法可用于定位作为参数给出的关键字的值。save 方法可用于把 Properties 对象的内容存放到作为第一个参数给出的 OutputStream 对象中。load 方法可从作为第一个参数给出的 InputStream 对象中恢复 Properties 对象的内容。propertyNames 方法可用于获得属性名的一个 Enumeration。
- 在 Random 类中，Java 提供了扩展的随机数生成方法。Random 类的不带参数的构造函数采用时间作为随机数产生器的种子，因此每次调用时的结果都不同。为了创建一个具有重复性特征的伪随机数产生器，我们采用带一个种子参数的 Random 构造函数。
- Random 类的 setSeed 方法设定种子值。nextInt 方法和 nextLong 方法可用于产生一致分布的随机整数。nextFloat 方法和 nextDouble 方法可用于产生  $0.0 \leq x < 1.0$  范围内的一致分布的值。
- 如果两个操作数相应的位都为 1，位与 (&) 运算符就把结果中的该位置为 1。
- 如果两个操作数相应的位中有一个 (或两个) 是 1，位或运算符 (|) 就把结果中的该位置为 1。
- 如果两个操作数相应的位中只有一个是 1，那么位异或运算符 (^) 就把结果中的该位置为 1。
- 左移运算符 (<<) 把它左操作数的位序列向左移动由右操作数指定的位数。
- 带符号扩展右移运算符 (>>) 把它左操作数的位序列向右移动由右操作数指定的位数——如果左操作数是负的，那么就把 1 序列从左边移入左操作数；否则，就将 0 序列从左边移入。
- 带零扩展右移运算符 (>>>) 把它左操作数的位序列向右移动由右操作数指定的位数——0 序列从左边移入左操作数。
- 位取反运算符 (~) 把操作数中的所有 0 变为结果中的 1，所有 1 变为结果中的 0。
- 每个位运算符 (位取反除外) 都有相应的赋值运算符。
- 不带参数的 BitSet 构造函数可创建一个空的 BitSet 对象。带有一个参数的 BitSet 构造函数则创建一个 BitSet 对象，其位数由它的参数指定。
- BitSet 的 set 方法可把指定位置为 “on”，clear 方法可把指定位置为 “off”。如果该位为 on，则 get 方法将返回 true；如果为 off，则返回 false。
- BitSet 的 and 方法可在 BitSet 对象之间执行一次按位的逻辑与，结果存放在由该方法激活的 BitSet 中。位逻辑或和位逻辑异或操作由 or 方法和 xor 方法执行。
- BitSet 的 size 方法可返回 BitSet 对象的大小。clone 方法可复制 BitSet 并返回一个指向新 BitSet 的 Object。toString 方法可把 BitSet 转换为 String。

## 术语

- addElement method of class Vector Vector 类的 addElement 方法
- addObserver method of Observable Observable 类的 addObserver 方法
- after method of class Date Date 类的 after 方法
- and method of class BitSet BitSet 类的 and 方法
- ArrayIndexOutOfBoundsException
- before method of class Date Date 类的 before 方法
- bit set 位集
- BitSet class BitSet 类
- capacity increment of a Vector Vector 对象容量的增量
- capacity of a Vector Vector 的容量
- capacity method of class Vector Vector 类的 capacity 方法
- clear method of class Hashtable Hashtable 类的 clear 方法
- clearChanged method of Observable Observable 类的 clearChanged 方法
- clone method of class BitSet BitSet 类的 clone 方法
- collision in hashing 散列冲突
- contains method of class Vector Vector 类的 contains 方法
- containsKey method of class Hashtable Hashtable 类的 containsKey 方法
- countObservers method of Observable Observable 类的 countObservers 方法
- Date class Date 类
- defaults 默认
- deleteObserver method of Observable Observable 类的 deleteObserver 方法
- deleteObservers method of Observable Observable 类的 deleteObservers 方法
- delimiter set 去限器集合
- Dictionary class Dictionary 类
- dynamically resizable array 大小动态可变的数组
- elementAt method of class Vector Vector 类的 elementAt 方法
- elements method of class Dictionary Dictionary 类的 elements 方法
- elements method of class Vector Vector 类的 elements 方法
- empty method of class Vector Vector 类的 empty 方法
- EmptyStackException class EmptyStackException 类
- enumerate successive elements 可枚举的连续元素
- Enumeration interface Enumeration 接口
- equals method of class Object Object 类的 equals 方法
- firstElement method of class Vector Vector 类的 firstElement 方法
- get method of class Dictionary Dictionary 类的 get 方法
- getDate method of class Date Date 类的 getDate 方法
- getDay method of class Date Date 类的 getDay 方法
- getHours method of class Date Date 类的 getHours 方法
- getMinutes method of class Date Date 类的 getMinutes 方法
- getMonth method of class Date Date 类的 getMonth 方法
- getProperty method of class Properties Properties 类的 getProperty 方法
- getSeconds method of class Date Date 类的 getSeconds 方法
- getTime method of class Date Date 类的 getTime 方法
- getYear method of class Date Date 类的 getYear 方法
- hasChanged method of class Observable Observable 类的 hasChanged 方法
- hashing 散列
- Hashtable class Hashtable 类
- hashCode method of class Object Object 类的 hashCode 方法
- hasMoreElements method of Enumeration Enumeration 类的 hasMoreElements 方法



- `IllegalArgumentException`  
`indexOf` method of class `Vector` `Vector`类的`indexOf`方法  
`initial capacity of a Vector` `Vector`对象的初始容量  
`insertElementAt` method of class `Vector` `Vector`类的`insertElementAt`方法  
`IOException`  
`isEmpty` method of class `Dictionary` `Dictionary`类的`isEmpty`方法  
`isEmpty` method of class `Vector` `Vector`类的`isEmpty`方法  
`iterator operations` 迭代操作  
`java.util`  
`key in a Dictionary` `Dictionary`中的关键字  
`keys` method of class `Dictionary` `Dictionary`类的`keys`方法  
`lastElement` method of class `Vector` `Vector`类的`lastElement`方法  
`lastIndexOf` method of class `Vector` `Vector`类的`lastIndexOf`方法  
`last-in-first-out( LIFO ) stack` 后进先出 ( LIFO ) 的堆栈  
`list` method of class `Properties` `Properties`类的`list`方法  
`load` method of class `Properties` `Properties`类的`load`方法  
`load factor in hashing` 散列中的负荷因子  
`nextDouble` method of class `Random` `Random`类的`nextDouble`方法  
`nextElement` method of class `Random` `Random`类的`nextElement`方法  
`nextFloat` method of class `Random` `Random`类的`nextFloat`方法  
`nextInt` method of class `Random` `Random`类的`nextInt`方法  
`nextLong` method of class `Random` `Random`类的`nextLong`方法  
`nextToken` method of class `Random` `Random`类的`nextToken`方法  
`NoSuchElementException` class  
`NoSuchElementException` 类  
`notifyObservers` method of class `Observable`  
`Observable` 类的`notifyObservers`方法  
`NullPointerException`  
`Observable` class `Observable` 类  
`observable list` `observable` 表  
`observer` 观察者  
`Observer` interface `Observer` 接口  
`or` method of class `BitSet` `BitSet`类的`or`方法  
`peek` method of class `Stack` `Stack`类的`peek`方法  
`pop` method of class `Stack` `Stack`类的`pop`方法  
`propertyNames` method of class `Properties` `Properties`类的`propertyNames`方法  
`Properties` class `Properties` 类  
`pseudo-random numbers` 伪随机数  
`push` method of class `Stack` `Stack`类的`push`方法  
`put` method of class `Dictionary` `Dictionary`类的`put`方法  
`Random` class `Random` 类  
`rehash` method of class `Hashtable` `Hashtable`类的`rehash`方法  
`remove` method of class `Dictionary` `Dictionary`类的`remove`方法  
`removeAllElements` method of class `Vector` `Vector`类的`removeAllElements`方法  
`removeElement` method of class `Vector` `Vector`类的`removeElement`方法  
`removeElementAt` method of class `Vector` `Vector`类的`removeElementAt`方法  
`save` method of class `Properties` `Properties`类的`save`方法  
`search` method of class `Stack` `Stack`类的`search`方法  
`seed of a random number generator` 随机数产生器的种子  
`set` method of class `BitSet` `BitSet`类的`set`方法  
`setChanged` method of class `Observable` `Observable`类的`setChanged`方法  
`setDate` method of class `Date` `Date`类的`setDate`方法  
`setElementAt` method of class `Vector` `Vector`类的`setElementAt`方法  
`setMinutes` method of class `Date` `Date`类的`setMinutes`方法

setMonth method of class Date Date类的setMonth方法

setSeconds method of class Date Date类的setSeconds方法

setSeed method of class Random Random类的setSeed方法

setSize method of class Vector Vector类的setSize方法

setTime method of class Date Date类的setTime方法

setYear method of class Date Date类的setYear方法

size method of class Dictionary Dictionary类的size方法

size method of class Vector Vector类的size方法

Stack class Stack类

toString method of class Date Date类的toString方法

trim a Vector to its exact size 把Vector对象缩小到与它实际的大小一样

trimToSize method of class Vector Vector类的trimToSize方法

update method of interface Observer Observer接口的update方法

Vector class Vector类

whitespace characters 空白字符

xor method of class BitSet BitSet类的xor方法

## 自测练习

### 18.1 填空：

- Java类提供了类似于数组的 \_\_\_\_\_ 数据结构，它可动态改变自己的大小。
- 如果没有指定增加的容量，那么每次在需要增加容量时系统会自动把Vector的大小变为原来的 \_\_\_\_\_。
- 如果存储空间很宝贵，那么可以使用Vector类的 \_\_\_\_\_ 方法来把Vector的大小缩减到与它实际的大小一样。

### 18.2 判断下面的句子是否正确。如果不正确，请解释原因。

- 原始数据类型的值可以直接存储在Vector中。
- 对于散列而言，如果负荷因子增大，那么系统冲突的机会就会减少。

### 18.3 在什么情况下程序会抛出EmptyStackException异常。

### 18.4 填空：

- 如果每个操作数中相应的位都是1，那么使用 \_\_\_\_\_ 运算符的表达式结果的该位将置为1。否则结果中的位将置为0。
- 如果每个操作数中相应的位至少有一个是1，那么使用 \_\_\_\_\_ 运算符的表达式结果的该位将置为1。否则结果的位将置为0。
- 如果每个操作数中相应的位只有一位是1，那么使用 \_\_\_\_\_ 运算符的表达式结果的该位将置为1。否则结果中的位将置为0。
- 位与运算符(&)经常用于 \_\_\_\_\_ 位，也就是通过把位序列中其余的位都置为0来选择某些位。
- \_\_\_\_\_ 运算符用于左移一个值的位序列。
- \_\_\_\_\_ 运算符用于带符号扩展右移一个值的位序列，\_\_\_\_\_ 运算符用于带零扩展右移一个值的位序列。

## 自测练习答案

- 18.1 a) Vector。b) 两倍。c) trimToSize。
- 18.2 a) 不正确。Vector 只存储 Object，必须使用 java.lang 软件包中的类型包装类（Integer、Long、Float、Double、Boolean 和 Character）来创建包含原始数据类型值的 Object。  
b) 不正确。随着负荷因子的增大，可用的关键字/值对的数目相对于整个关键字/值对的数目而言会越来越少，因此，使用散列操作选择一个已经被占用的关键字/值对（冲突）的机会将会增加。
- 18.3 对一个空的 Stack 对象执行 pop 操作或对一个空的 Stack 对象执行 peek 操作。
- 18.4 a) 位与（&）。b) 位或（|）。c) 位异或（^）。d) 掩盖。e) 左移运算符（<<）。f) 带符号扩展右移运算符（>>）。g) 带零扩展右移运算符（>>>）。

## 练习

- 18.5 使用与散列有关的内容来描述下面几项的定义。
- a) 应用程序关键字
  - b) 冲突
  - c) 散列变换
  - d) 负荷因子
  - e) 时间、空间的交换
  - f) 散列表的容量
- 18.6 简要解释下面给出的 Vector 类的方法所完成的操作：
- a) addElement
  - b) insertElement
  - c) setElementAt
  - d) removeElement
  - e) removeAllElements
  - f) removeElementAt
  - g) firstElement
  - h) lastElement
  - i) isEmpty
  - j) contains
  - k) indexOf
  - l) trimToSize
  - m) size
  - n) capacity
- 18.7 解释为什么向一个当前规模小于 Vector 对象容量的 Vector 对象中插入一个元素是一个比较快的操作，而向一个当前规模等于容量的 Vector 对象中插入一个元素是一个比较慢的操作。
- 18.8 在前面我们已经指出，按照默认值一次增加 Vector 容量一倍的方法看起来有些浪费空间，

但是对许多快速增加的 Vector 来说,这样做恰好是能够“接近要求”的有效方法。请解释原因,说明这种加倍算法的优点和缺点。如果一个程序判断出加倍算法正在浪费存储空间,那么应该怎么办?

- 18.9 使用 Vector 类的对象说明 Enumeration 接口的用法:
- 18.10 通过对 Vector 类的扩展,Java 的设计者们能够很快得到 Stack 类。这种用继承来构造类的方法的缺点是什么(特别对于 Stack 类)?
- 18.11 简要解释下面给出的 Hashtable 类的方法所完成的操作:
- a) put
  - b) get
  - c) remove
  - d) isEmpty
  - e) containsKey
  - f) contains
  - g) clear
  - h) elements
  - i) keys
- 18.12 说明如何使用 Random 类来创建我们在调试中需要的具有重复性的伪随机数。
- 18.13 通过重用 Date 类,用户可以不用编写自己的日期处理代码,这样可以节省时间并且避免代价巨大的错误。请研究 Date 类的方法,熟悉该类的用法。编写一个演示 Date 类所有方法的 Java applet,以教会别人如何使用 Date 类。
- 18.14 说明“Observer 和 Observable 对象”设计模板的用法。编写出现实世界中具有这种相互作用的 5 个例子。说明下面给出的方法的操作,这些方法可以完成 Observer/Observable 的相互作用。
- a) addObserver
  - b) deleteObserver
  - c) notifyObservers
  - d) deleteObservers
  - e) setChanged
  - f) clearChanged
  - g) hasChanged
  - h) countObservers
- 18.15 我们指出,一个 Properties 对象是一个“永久的”Hashtable 对象,这句话的意思是什么?说明下面给出的 Properties 类的方法所完成的操作:
- a) load
  - b) save
  - c) getProperty
  - d) propertyNames
  - e) list
- 18.16 为什么要使用 BitSet 类的对象?说明下面给出的 BitSet 类的方法所完成的操作:
- a) set
  - b) clear

- e) get
- d) and
- e) or
- f) xor
- g) size
- h) equals
- i) clone
- j) toString
- k) hashCode

- 18.17 编写一个程序，带符号扩展右移一个整型变量 4 位，然后带零扩展右移同一个整型变量 4 位。该程序应该在每次移位操作之前和之后均打印出该整数的位序列表示。使用一个正整数运行程序一次，然后用一个负整数再运行一次。
- 18.18 说明如何使用左移一个整数 1 位来模拟乘 2 的操作，以及如何通过右移一个整数 1 位来模拟除 2 的操作。注意考虑与该整数的符号有关的问题。
- 18.19 编写一个程序，颠倒一个整数的位序列的顺序。该程序应该由用户输入值，并调用方法 `reverseBits` 来打印颠倒顺序后的位序列。在位序列颠倒之前和之后都打印该数的位序列，以确保正确地执行了操作。读者既可以使用递归的方法也可以使用迭代的方法来实现该程序。

## 附录 A 运算符优先级表

下列运算符自上而下按照优先级递减的顺序排列。

运算符	类型	结合顺序
()	括号	从左到右
[]	数组 成员 选择	下标
++	一元后置递增	从右到左
--	一元后置递减	
++	一元前置递增	
--	一元前置递减	
+	一元加	
-	一元减	
!	一元逻辑非	
~	一元按位取反	
(type)	一元强制类型转换	
*	乘	从左到右
/	除	
%	取模	
+	加	从左到右
-	减	
<<	位左移	从左到右
>>	带符号扩展右移	
>>>	带零扩展右移	
<	小于关系	从左到右
<=	小于等于关系	
>	大于关系	
>=	大于等于关系	
instanceof	类型比较	
==	相等关系	从左到右
!=	不相等关系	
&	按位与	从左到右
^	按位异或 布尔逻辑异或	从左到右
	按位或 布尔逻辑或	从左到右
&&	逻辑与	从左到右
	逻辑或	从左到右
?:	三元运算符	从右到左
=	赋值	从右到左
+=	赋值加	
-=	赋值减	
*=	赋值乘	
/=	赋值除	
%=	赋值取模	

(续表)		
运算符	类型	结合顺序
&=	赋值按位与	从左到右
^=	赋值按位异或	
=	赋值按位或	
<<=	赋值按位左移	
>>=	赋值带符号扩展右移	
>>>=	赋值带零扩展右移	

图 A.1 运算符优先级表

## 附录 B ASCII 字符集

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

该表左侧的数字为字符编码十进制值(0~127)的高位,表上边的数字是字符编码十进制数值的低位。例如,字符'F'的代码为70,字符'&'的代码为38。



## 附录C 数值系统

### 学习目标

- 理解基本数值系统的概念例如基数、位值、符号值
- 理解如何运用以二进制、八进制和十六进制数值系统表示的数值
- 能够把二进制数简化为八进制或十六进制数
- 能够将八进制数和十六进制数转换为二进制数
- 能够进行十进制与二进制、八进制、十六进制数之间的转换
- 理解二进制算术运算及如何用补码表示负的二进制数

### C.1 简介

本附录介绍了Java程序员使用的重要的数值系统，特别是他们开发的项目要求不与“机器级”硬件交互时更是如此。这样的项目包括操作系统、计算机网络软件、编译器、数据库系统及各种高性能要求的应用程序。

当我们在Java程序中用到227或-63等整数时，我们假定了这些数值是以十进制数值系统（以10为基数）表示的。十进制数值系统中的数码有0、1、2、3、4、5、6、7、8和9。最小的数码为0，最大数码为9（比基数10小1）。计算机内部使用的是二进制数值系统（基数为2）。二进制数值系统只有两个数码，即0和1，最小数码为0，最大数码为1（比基数2小1）。

可以看到，二进制数比等值的十进制数长得多。汇编语言和像C这样的高级语言可以深入到“机器级”编程。但使用这些语言的程序员发现二进制数用起来很麻烦。因此，另外两种数值系统——八进制数值系统（基数为8）和十六进制数值系统（基数为16）因为能够方便地简化二进制数而受到了人们的欢迎。

八进制数值系统的数码范围为0~7。因为二进制数值系统和八进制数值系统都比十进制数值系统的数码少，所以它们的数码与相应的十进制数码相同。

而十六进制数值系统需要十六个数码——最小数码为0，最大数码的值相当于十进制数15（比基数16小1）。习惯上，我们用字母A到F表示对应于十进制数10~15的十六进制数码。这样在十六进制中我们可能得到仅含十进制数码的十六进制数（如876），也可得到同时包含数码和字母的数值（如8A55F）以及只含字母的数值（如FFE）。有时，十六进制数刚好拼成单词（如FACE、FEED），这常使程序员感到别扭。

以上每种数值系统都使用按位记数法——每个数位有一个不同的位值。例如：对于十进制数937（9、3、7称为符号值），我们说7在个位、3在十位、9在百位。注意，每个数位实际是一个基数的乘幂（基数为10），其乘幂从右到左依次递增，从0开始，每次加1（见图C.3）。

再长的十进制数，后面将是千位（10的3次乘幂）、万位（10的4次乘幂）、十万位（10的5次乘幂）、百万位（10的6次乘幂）、千万位（10的7次乘幂），依次类推。

对于二进制数101，我们说最右边的1在一位，0在二位，最左边的1在四位。注意，每个数

位实际是一个基数 2 的乘幂，乘幂从右到左依次递增，从 0 开始，每次加 1（见图 C.4）。

再长的二进制数，后面将是八位（2 的 3 次乘幂）、十六位（2 的 4 次乘幂）、三十二位（2 的 5 次乘幂）、六十四位（2 的 6 次乘幂），依次类推。

对八进制数 425，我们说 5 在一位，2 在八位，4 在六十四位。每个数位实际是基数 8 的乘幂，乘幂从右到左依次递增，从 0 开始，每次加 1（见图 C.5）。

再长的八进制数，后面将是五百二十位（8 的 4 次乘幂）、三万二千七百六十八位（8 的 5 次乘幂），依次类推。

对十六进制数 3DA，我们说 A 在个位，D 在十六位，3 在二百五十六位。每个数位实际是基数 16 的乘幂，乘幂从右到左依次递增，从 0 开始，每次加 1（见图 C.6）。

再长的十六进制数，后面将是四千零九十六位（16 的 3 次乘幂）、三万二千七百六十八位（16 的 4 次乘幂），依次类推。

二进制数码	八进制数码	十进制数码	十六进制数码
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A（十进制值 10）
			B（十进制值 11）
			C（十进制值 12）
			D（十进制值 13）
			E（十进制值 14）
			F（十进制值 15）

图 C.1 二进制、八进制、十进制和十六进制数值系统的数码

属性	二进制	八进制	十进制	十六进制
基数	2	8	10	16
最小数码	0	0	0	0
最大数码	1	7	9	F

图 C.2 二进制、八进制、十进制和十六进制数值系统的比较

十进制数值系统中的位值			
十进制数码	9	3	7
数位名	百位	十位	个位
位值	100	10	1
位值的乘幂形式	10 <sup>2</sup>	10 <sup>1</sup>	10 <sup>0</sup>
基数（10）			

图 C.3 十进制数值系统中的位值

二进制数值系统中的位值			
二进制数码	1	0	1
数位名	四位	二位	一位
位值	4	2	1
位值的乘幂形式	$2^2$	$2^1$	$2^0$
基数 (2)			

图 C.4 二进制数值系统中的位值

八进制数值系统中的位值			
八进制数码	4	2	5
数位名	八十四位	八位	一位
位值	64	8	1
位值的乘幂形式	$8^2$	$8^1$	$8^0$
基数 (8)			

图 C.5 八进制数值系统中的位值

十六进制数值系统中的位值			
十六进制数码	3	D	A
数位名	二百五十六位	十六位	一位
位值	256	16	1
位值的乘幂形式	$16^2$	$16^1$	$16^0$
基数 (16)			

图 C.6 十六进制数值系统中的位值

## C.2 将二进制数简化为八进制和十六进制数

八进制和十六进制数在计算的主要用途是简化冗长的二进制数。图 C.7 表明冗长的二进制数可用基数更高的数值系统简洁地表达。

十进制	二进制	八进制	十六进制
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

图 C.7 等值的十进制、二进制、八进制与十六进制数

八进制、十六进制系统与二进制系统有一个特别重要的关系，就是八进制与十六进制的基数（8和16）是二进制基数2的乘幂。考察下面的12位二进制数以及与之相等的八进制和十六进制数，看看这种关系是如何方便地把二进制数简化为八进制和十六进制数的。

二进制数	八进制数	十六进制数
100011010001	4321	8D1

二进制转换为八进制，只需将12位二进制数按每三个连续位分为一组，并按以下方式将每组写成八进制数：

100	011	010	001
4	3	2	1

注意，在每组下面的八进制数刚好与对应这三位的二进制数相等（见图C.7）。

在从二进制到十六进制的转换中，我们可以发现同样的关系。它是将12位二进制数按照每4个连续位分成一组，并按以下方式将每组写成十六进制数：

1000	1101	0001
8	D	1

可以看到，在每组下面的十六进制数刚好与对应这四位的二进制数相等（见图C.7）。

C.3 将八进制和十六进制数转换为二进制数

上一节中，我们知道了如何将二进制数转换为等值的八进制和十六进制数，就是通过把二进制数位分组，然后简单地代之以等值的八进制或十六进制值。这个处理反过来也可以用于产生与给定八进制或十六进制数相等的二进制数。

例如，要将八进制数653转换为二进制数，只需将6、5和3分别写成与之相等的3位二进制数110、101和011，这就形成了与八进制653相等的9位二进制数110101011。

要将十六进制数FAD5转换为二进制数，只需将F、A、D和5分别写成与之相等的4位二进制数1111、1010、1101和0101，这就形成了与十六进制数FAD5相等的16位二进制数1111101011010101。

C.4 将二进制、八进制和十六进制数转换为十进制数

因为我们已习惯于十进制，所以经常将二进制、八进制、十六进制数转换为十进制，从而找到“真实”值的感觉。C.1节中的图表示了十进制的位值。从其他进制转换为十进制，只需将每个数码的十进制数乘以其位值，然后求出这些积的和。例如，图C.8把二进制数110101转换成了十进制数53。

将二进制数转换为十进制数						
位值	32	16	8	4	2	1
符号值	1	1	0	1	0	1
乘积	1* 32=32	1* 16=16	0* 8=0	1* 4=4	0* 2=0	1* 1=1
总和	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

图 C.8 将二进制数转换为十进制数

我们还可以使用同样的办法将八进制数7614和十六进制数AD3B转换为十进制数3980和44347（见图 C.9 和图 C.10）。

将八进制数转换为十进制数				
位值	512	64	8	1
符号值	7	6	1	4
乘积	$7 \times 512 = 3584$	$6 \times 64 = 384$	$1 \times 8 = 8$	$4 \times 1 = 4$
总和	$= 3584 + 384 + 8 + 4 = 3980$			

图 C.9 将八进制数转换为十进制数

将十六进制数转换为十进制数				
位值	4096	256	16	1
符号值	A	D	3	B
乘积	$A \times 4096 = 40960$	$D \times 256 = 3328$	$3 \times 16 = 48$	$B \times 1 = 11$
总和	$= 40960 + 3328 + 48 + 11 = 44347$			

图 C.10 将十六进制数转换为十进制数

C.5 将十进制数转换为二进制、八进制或十六进制数

上一节中的转换是根据常规的按位记数法进行的。从十进制到二进制、八进制、十六进制的转换也遵循该规则。

例如将十进制数 57 转换为二进制。我们从右到左写出每列的位值，直到发现位值大于该十进制数的列。我们不需要该列，所以将之丢弃。这样首先得到：

位值: 64 32 16 8 4 2 1

然后，去掉位值为 64 的列，得到：

位值: 32 16 8 4 2 1

下一步，我们从左到右进行。先用 57 除以 32，得到商为 1，余数是 25，所以我们在 32 这一列下写上 1。然后用 25 除以 16，得到商为 1，余数是 9，所以我们在 16 这一列下写上 1。再用 9 除以 8，得到商为 1，余数是 1，所以我们在 8 这一列上写上 1。其后两列我们得到商为 0，所以写上两个 0。最后，1 除以 1 得 1，写上 1。这样就得到了下面的结果：

位值: 32 16 8 4 2 1  
符号值: 1 1 1 0 0 1

因此，十进制数 57 等于二进制数 111001。

将十进制数 103 转换成八进制。我们从右到左写每列的位值，直到发现位值大于该十进制数的列。我们不需要该列，所以将之丢弃。这样首先得到：

位值: 512 64 8 1

然后，我们去掉位值为 512 的列，得到：

位值: 64 8 1

下一步从左到右进行。用 103 除以 64，得到商为 1，余数是 39，所以在 64 这一列下写上 1。用

39 除以 8，得到商为 4，余数是 7，所以在 8 这一列下写上 4。最后，7 除以 1 得 1，没有余数，写上 7。结果如下所示：

```
位值: 64 8 1
符号值: 1 4 7
```

因此，十进制数 103 等于八进制数 147。

将十进制数 375 转换成十六进制。我们从右到左写每列的位值，直到发现位值大于该十进制数的列。我们不需要该列，所以将之丢弃。这样首先得到：

```
位值 4096 256 16 1
```

然后，我们去掉位值为 4096 的列，得到：

```
位值 256 16 1
```

下一步从左到右进行。用 375 除以 256，得到商为 1，余数是 119，所以在 256 这一列下写上 1。用 119 除以 16，得到商为 7，余数是 7，所以在 16 这一列下写上 7。最后，7 除以 1，没有余数，写上 7。结果如下所示：

```
位值: 256 16 1
符号值: 1 7 7
```

因此，十进制数 375 等于十六进制数 177。

## C.6 负的二进制数：补码表示法

本附录中的讨论都集中在正数上。这一节，我们解释计算机是如何用补码表示法表示负数。首先解释补码如何形成一个二进制数，然后看看为什么它能表示给定二进制数的负数。

考察一个 32 位整数的机器。假定：

```
int value = 13;
```

value 的 32 位表达式是：

```
00000000 00000000 00000000 00001101
```

为了形成 value 的负值，我们首先利用 Java 的按位取反运算符(~)形成它的反码：

```
ones_Complement of Value= ~value
```

在机器内部，~value 现在是对每个数位取反后的 value，即 0 变成 1，1 变成 0：

```
value:
00000000 00000000 00000000 00001101

~value (即 value 的反码):
11111111 11111111 11111111 11110010
```

要形成 value 的补码，我们只需将 value 的反码加 1。即：

```
value 的补码:
11111111 11111111 11111111 11110011
```

如果它确实就等于-13，那么它与13的二进制数相加就应该得到结果0，如下所示：

00000000	00000000	00000000	00001101
+11111111	11111111	11111111	11110011
00000000	00000000	00000000	00000000

最左边的进位被丢弃，事实上我们得到了结果0。如果我们将一个数的反码加上这个数，结果将是所有位全部为1。要得到全0的方法就是用反码加1得到补码。加1导致每位变为0并进1。进位一直左移直到被最左位丢弃，因此结果为全0。

计算机执行减法：

```
x = a - value;
```

实际上是执行a加value的补码：

```
x = a + (~value + 1);
```

假定a为27，value为13。如果value的补码确实是value的负数，那么value的补码加a的结果应该是14，如下所示：

a (=27)	00000000	00000000	00000000	00011011
+ (~value+1)	+ 11111111	11111111	11111111	11110011
	00000000	00000000	00000000	00001110

结果的确等于14。

## 小结

- 当我们在Java程序中用到227或-63等整数时，这些值是自动假定是以十进制数值系统（以10为基数）表示的。十进制数值系统中的数码有0、1、2、3、4、5、6、7、8和9。最小的数码为0，最大数码为9（比基数10小1）。
- 计算机内部使用的是二进制数值系统（基数为2）。二进制数值系统只有两个数码，即0和1，最小数码为0，最大数码为1（比基数2小1）。
- 因为八进制（基数为8）和十六进制（基数为16）数值系统可方便地简化二进制数，所以得到了广泛的应用。
- 八进制数值系统的数码从0~7。
- 十六进制数值系统需要十六个数码，最小数码为0，最大数码的值相当于十进制数15（比基数16小1）。习惯上，我们用字母A到F表示对应于十进制数10~15的十六进制数码。
- 每一种数值系统都使用了按位记数法。数码所在的每一位有不同的位值。
- 八进制、十六进制数值系统与十进制数值系统之间有一个特别重要关系，就是八进制和十六进制的基数（8和16）是二进制基数（2）的乘幂。
- 只需将八进制数的每个数位用相等的三位二进制数表示，就可以把八进制数转换为二进制数。
- 只需将十六进制数的每个数位用相等的四位二进制数表示，就可以把十六进制数转换为二进制数。
- 因为我们习惯于十进制，所以经常将二进制、八进制或十六进制数转换为十进制，以获得“真实”值的感觉。

- 从其他进制转换为十进制, 只需将每个数码的十进制数乘以其位值, 然后求出这些积的和。
- 计算机是用补码表示法表示负数。
- 在二进制中, 要形成一个值的负值, 首先用C语言的按位取反运算符(~)形成它的反码(使得该值的所有位取反), 然后只需将该值的反码加1。

## 术语

base 基数	digit 数码
base 2 number system 以2为基数的数值系统	hexadecimal number system 十六进制数值系统
base 8 number system 以8为基数的数值系统	negative value 负值
base 10 number system 以10为基数的数值系统	octal number system 八进制数值系统
base 16 number system 以16为基数的数值系统	one's complement notation 反码表示法
binary number system 二进制数值系统	positional notation 按位记数法
bitwise complement operator(~) 按位取反运算符(~)	positional value 位值
conversions 转换	symbol value 符号值
decimal number system 十进制数值系统	two's complement notation 补码表示法

## 自测练习

- C.1 十进制、二进制、八进制和十六进制数值系统的基数分别是\_\_\_\_、\_\_\_\_、\_\_\_\_和\_\_\_\_。
- C.2 (选择) 通常, 一个给定二进制数的十进制、八进制和十六进制表示法含有比二进制更(多/少)的数位。
- C.3 (判断正误) 使用十进制数值系统的通常原因是它通过将四位二进制组替换为十进制数位, 可以简化二进制数。
- C.4 (选择) 一个非常大的二进制数的(八进制/十六进制/十进制)表示法是最简洁的。
- C.5 (判断正误) 任何进制中的最大数码都是比基数大1。
- C.6 (判断正误) 任何进制中的最小数码都是比基数小1。
- C.7 不论是在二进制、八进制、十进制还是在十六进制中, 任何数的最右边数位的位值通常等于\_\_\_\_\_。
- C.8 不论是在二进制、八进制、十进制还是在十六进制中, 任何数的次右边数位的位值通常等于\_\_\_\_\_。
- C.9 补齐以下表示各数值系统的右边4个位值。

十进制	1000	100	10	1
十六进制	...	256	...	...
二进制	...	...	...	...
八进制	512	...	8	...

- C.10 将二进制数 110101011000 转换为八进制数和十六进制数。
- C.11 将十六进制数 FACE 转换为二进制数。



- C.12 将八进制数 7316 转换为二进制数。  
 C.13 将十六进制数 4FEC 转换为八进制数。提示：先转换为二进制数。  
 C.14 将二进制数 1101110 转换为十进制数。  
 C.15 将八进制数 317 转换为十进制数。  
 C.16 将十六进制数 EFD4 转换为十进制数。  
 C.17 将十进制数 177 转换为二进制、八进制和十六进制数。  
 C.18 写出把十进制数 417 表示为二进制数的过程。然后写出其反码和补码。  
 C.19 一个数的补码加上其本身会是什么结果？

## 自测练习答案

- C.1 10、2、8、16  
 C.2 少。  
 C.3 不正确。  
 C.4 十六进制。  
 C.5 不正确。任何进制中的最大数码都是比基数小 1。  
 C.6 不正确。任何进制中的最小数码都是 0。  
 C.7 1 (基数的 0 次幂)。  
 C.8 该数值系统的基数。
- |     |      |      |     |    |   |
|-----|------|------|-----|----|---|
| C.9 | 十进制  | 1000 | 100 | 10 | 1 |
|     | 十六进制 | 4096 | 256 | 16 | 1 |
|     | 二进制  | 8    | 4   | 2  | 1 |
|     | 八进制  | 512  | 64  | 8  | 1 |
- C.10 八进制数 6530；十六进制数 D58。  
 C.11 二进制数 1111 1010 1100 1110。  
 C.12 二进制数 111 011 001 110。  
 C.13 二进制数 0 100 111 111 101 100；八进制数 47754。  
 C.14 十进制数  $2+4+8+32+64=110$ 。  
 C.15 十进制数  $7+1*8+3*64=7+8+192=207$ 。  
 C.16 十进制数  $4+13*16+15*256+14*4096=61396$ 。  
 C.17 十进制数 177 转换成二进制数：
- |                                                 |     |    |    |    |   |   |   |   |
|-------------------------------------------------|-----|----|----|----|---|---|---|---|
| 256                                             | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 128                                             | 64  | 32 | 16 | 8  | 4 | 2 | 1 |   |
| $(1*128)+(0*64)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)$ |     |    |    |    |   |   |   |   |
| 10110001                                        |     |    |    |    |   |   |   |   |
- 转换成八进制数：
- |                      |    |   |   |
|----------------------|----|---|---|
| 512                  | 64 | 8 | 1 |
| 64                   | 8  | 1 |   |
| $(2*64)+(6*8)+(1*1)$ |    |   |   |
| 261                  |    |   |   |
- 转换成十六进制数：
- |                 |    |   |
|-----------------|----|---|
| 256             | 16 | 1 |
| 16              | 1  |   |
| $(11*16)+(1*1)$ |    |   |

$$(B \times 6) + (1 \times 1)$$

B1

C.18 二进制:

```

512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1)
110100001

```

反码: 001011110

补码: 001011111

检验: 原二进制数加它的补码。

```

110100001
001011111
-----
000000000

```

C.19 为 0。

## 练习

C.20 有人说在十二进制数值系统中我们的许多计算会更容易, 因为 12 比十进制的基数 10 更具有可除性。在十二进制中, 什么是最小数码? 什么可能是十二进制数位的最高表示符? 其最右边四位的位值是多少?

C.21 任何数值系统中数的次右边数位的位值相对应的最高符号值是什么?

C.22 补齐以下各数值系统右边的 4 个位值。

十进制	1000	100	10	1
6 进制	...	...	6	...
13 进制	...	169	...	...
3 进制	27	...	...	...

C.23 将二进制数 100101111010 转换成八进制和十六进制数。

C.24 将十六进制数 3A7D 转换成二进制数。

C.25 将十六进制数 765F 转换成八进制数。提示: 先转换成二进制。

C.26 将二进制数 1011110 转换成十进制数。

C.27 将八进制数 426 转换成十进制数。

C.28 将十六进制 FFFF 转换成十进制数。

C.29 将十进制数 299 转换成二进制、八进制、十六进制数。

C.30 写出把十进制数 779 转换成二进制数的过程。然后写出其反码和补码。

C.31 一个数的补码加上其本身会是什么结果?

C.32 写出在 32 位整数的机器上整数值 -1 的补码。

## 附录 D 面向对象的电梯模拟程序

### 教学目标

- 介绍面向对象的基本元素,使得学生可以在学习计算机科学和软件工程课程的开始就能够参与课程设计
- 向学生介绍面向对象设计过程的基本要素
- 使得学生可以参与一个重要应用程序的设计
- 在这个重要的课程设计中,学生将使用到 Java 的一些更加高级的特性,包括图形处理、图形用户界面、异常处理、多线程,以及包括声音、图像和动画的多媒体处理
- 对于能力更强的学生提供大量关于本系统的修改建议

### D.1 简介

在本附录中,我们将进行一个完整的面向对象设计(OOD),使用面向对象编程(OOP)的技术来实现一个电梯模拟程序。这样的要求看起来有点复杂,但是我们一次只处理这个问题的一小部分,从而分步完成整个任务。电梯模拟程序是一个相当大的项目,它更适合作为一项课程设计的题目,或者由一个项目小组共同完成。

首先简单介绍面向对象的技术和概念,我们从介绍一些面向对象技术的关键概念开始。请观察一下你所在的现实世界,无论在什么地方都可以看到它们——对象(客观事物)!人类、动物、植物、计算机、汽车、建筑物、书籍、飞机、剪草机、家具,等等。人类采用唯物的方式进行思维。同时我们具有很强的抽象思维能力,它使得我们可以在屏幕上看到由不同颜色的亮点组成的图形时,自然地想起相应的客观事物(对象)——山脉、树木、直升飞机、云彩,等等。如果我们愿意,我们可以从一粒粒的沙子想到沙滩,从树木想到森林,从砖块想到房屋。

我们可能习惯于将事物对象分成两类——活动的对象和非活动的对象。活动的对象从某种角度来看是“活的”,它们可以四处移动并做些事情。非活动的对象,例如石头,看起来好像什么事情也不做,它们仅仅是“待在那里”。尽管如此,所有这些对象都拥有一些共性。它们都拥有诸如大小、形状、颜色、质量等这样的属性。并且,它们可以表现出不同的行为,例如一个球可以滚动、弹跳、充气以及放气;一个婴儿可以睡眠、哭闹、爬行、行走、流口水、眨眼睛;一辆汽车可以加速、刹车、转弯。

人类通过研究和观察不同事物的属性以及它们的行为表现来学习有关它们的概念。不同的事物对象可以拥有许多相同的属性,并且表现出类似的行为。我们可以进行一些比较,例如婴儿和成年人之间,人类和大猩猩之间拥有许多相同点。小汽车、卡车、红色的四轮马车、溜冰鞋之间都存在许多相同点。

面向对象的编程技术采用软件的对照物来模仿现实世界中的对象。面向对象的编程利用了类这样的关系——同一类的不同对象,例如一个运输工具类中的不同运输工具,它们都拥有相同的特性。面向对象的编程还利用了继承这样的关系——如果一个新创建的类是从现有类中继承面来的,那么

它们将拥有一部分相同的特性。如果我们将敞篷汽车类看做从汽车类继承的一个类,那么它们就拥有一部分相同的特性,但不同的是敞篷汽车的车顶可以上下移动。

面向对象的编程技术给我们带来的是以一个更加自然、更加直观的方式对程序设计过程进行观察,这也称为对现实世界的事物以及它们的属性和表现进行模拟。OOP同时还模仿了对象之间的通信,就像人们之间相互交流一样(例如一个人站在街角,发出手势要打一辆计程车),对象之间也可以利用消息进行通信。

OOP将数据(属性)和方法(表现)封装进称为对象的单元中,一个对象的数据和方法紧密地结合在一起。对象拥有信息隐藏的特性,这表明尽管通过定义良好的接口,对象之间可以相互通信,但是通常对象并不了解与之通信的一方具体是如何实现的——实现的细节由对象本身隐藏起来。其实这也很正常,我们驾驶一辆汽车并不需要了解有关它的引擎、发电机、电池、传输系统和损耗系统是如何工作的。我们将看到为什么信息隐藏对于良好的软件工程设计是非常重要的。

在C和其他过程式编程语言中,编程可以看成是面向行为的;而在Java中,编程则是面向对象的。在C语言中,程序的基本单元是函数;在Java中,程序的基本单元为类,对象从中进行实例化(即创建)。

C程序员更加关心如何编写合格的函数。执行某些特定操作的动作将作为函数组织起来,而程序就由各种不同的函数构成。数据在C中是相当重要的,但是从某种角度来看,数据仅仅是用来支持在程序中正常执行各种动作。一个系统说明文档中的谓词可以帮助C程序员进行判断,实现一个系统的函数集是否工作正常。

Java程序员则更加关心如何创建他们自己的、称为类的用户自定义类型。每一个类在包含数据的同时还包含操作这些数据的方法集。一个类中的数据成员称为实例变量。例如将一个内嵌的int类型称为变量,一个用户自定义类型(即一个类)则称为一个对象。程序员使用内嵌的数据类型作为构造用户自定义类型的基本模块。在Java中,我们关注的是对象而非其中的方法。一个系统说明文档中的名词用来帮助Java程序员进行判断,实现一个系统的对象集是否工作正常。

## D.2 问题陈述

某个公司需要建造一座两层的办公建筑物,并且在其中采用“最新”的电梯技术。这个公司要求读者开发一个Java applet来模拟电梯的操作,以判断这样的技术是否合乎他们的要求。

这个电梯可以用来搭乘一个人,它按照节约能源的要求进行设计,因此它仅在需要的时候才进行移动。电梯在白天启动的时候关着门,并且停在建筑物的第一层。这部电梯当然会不断变换运行方向——首先上升,然后下降。

模拟程序拥有一个简单的图形用户界面——包括一个单独“Create Person”按钮的applet。每当点击这个按钮时,模拟程序将创建一个“新”人,然后将他放在某一层上(即第一层或第二层),这个人想要到达的楼层由程序随机产生。然后这个人按下这一层的呼叫按钮,他想要到达的目的楼层应该同他现在所在的楼层不同。

如果当天第一个到达的人出现在第一层,他就可以立即搭乘电梯(当然在按下呼叫按钮并且等到楼门和电梯门开启之后)。如果第一个人想到第二层,那么电梯应该行进到第二层同时搭乘上这个人。在理想状态下,我们认为电梯在楼层之间的移动不消耗时间。在将来的版本中,要考虑更加现实的情况。

电梯到达某一楼层后将打开此层的一个指示灯,并使电梯响铃。楼层按钮和电梯按钮将重新设置,电梯门打开时,楼门也将打开,乘客从电梯中下来。如果一个乘客正好等在此层,他就可以进

入电梯并按下目的楼层的按钮。如果电梯需要移动,它将自己判断应去的方向(对于一部只停两层的电梯来说这并不困难),然后开始行进到另一层。为了简化,我们假设从电梯到达某一楼层直到这一层的楼门关闭之间的耗费时间为零。电梯总是知道它所处的楼层和想要到达的楼层。

由于在本模拟程序版本中所有事件的耗时都为零,因此一次只能够模拟一个人。在模拟过程中,模拟程序可以在下一个人到达之前处理完现有人的事物。换言之,每当创建一个人时,此人应按下呼叫按钮,搭乘上电梯,移动到另一个楼层,然后在另一个人进入某一楼层并等待电梯之前离开电梯。

## D.3 电梯实验室练习 1

(预备知识:第1章~第3章)

在本练习以及接下来的练习中,读者将分别体验面向对象设计的几个步骤。第一步是确立本问题中的类,需要采用正规的方式来实现这些类。在本练习中,首先应执行下列步骤:

1. 标识本模拟程序中用到的类。上一节的问题陈述中提到许多类的对象在一起工作,还有模拟电梯以及它同乘客的关系,楼层、电梯铃,等等。分析问题陈述中提到的名词,它们很可能代表了实现这个电梯模拟程序所必需的大多数类。
2. 对于所标识的每个类,应该按照简洁的原则给出它们的类名。类名应该反映它们在问题陈述中代表的事物。

### 注意

1. 这是一个很好的小组练习,读者可以参加一个两到四人的小组。你和你的同伴应该共同努力,应付挑战并且相互推敲对方的设计和实现。
2. 读者的小组应该同一个班级中的其他小组展开竞争,从而开发出“最好”的设计和实现。
3. 在第4章中,我们讨论了如何使用随机数产生器。每当模拟程序使用者按下“Create Person”按钮时,应使用 static Math 类的方法 random 来随机选择一个人所到的楼层。
4. 我们已经进行了大量简化问题的假设,你可以选择并完善它们的细节。
5. 由于真实世界确实是由各种不同的“对象”组成,即便读者从来没有正式地学习过面向对象的概念,也应该很自然地按照这样的思路完成这一课题。
6. 不要担心完美性,系统设计并非追求完美的过程,因此在设计时应该基于最大努力实施这个项目。
7. Java是个需要创造性的语言,因此不要过分依赖这里阐述的各种基本要求,可以在这个电梯模拟程序中加入自己的想像力。

## D.4 电梯实验室练习 2

(预备知识:第4章)

在前一个练习中,对于这个电梯模拟程序,我们开始了面向对象设计的第一步,也就是首先标识出为了实现这个项目所需的类。作为一个起点,鼓励大家列出问题陈述中相关的名词。在这一过程中,你将发现模拟程序中用到的类将包括电梯本身、人、楼层、建筑物、不同的按钮等。

类拥有属性和行为。类的属性在 Java 程序中通过数据进行表现,类的行为通过方法进行表现。在本练习中,我们的注意力集中在确定实现模拟程序所需的不同类的属性。在下面的练习中,我们将关注行为。在练习 4 中,我们将关注这个电梯模拟程序中对象之间的关系。

让我们开始练习之前讨论一下真实世界的各种属性。一个人的属性包括身高和体重。一台收音机的属性包括它的波段是 AM 还是 FM,当前收听的电台,当前音量的大小。一辆汽车的属性包括速度计和里程计指数。一座房屋的属性包括风格、房间数目、占地面积,以及其他的规格设置。一台计算机的属性包括制造商 (Apple、Compaq、Digital、IBM、Sun 等)、显示器的类型 (单色或彩色)、主存大小 (以百万字节计)、硬盘大小 (以百万字节或 10 亿字节计) 等。

1. 在开始工作时,使用一个字处理程序或编辑程序来输入“电梯实验室练习”中关于电梯模拟程序的问题陈述。
2. 将问题的各个事件进行展开。除去所有不相关的内容,然后将每个事件在文件中单独写成一行 (这里可能存在几十个问题陈述)。下面列出了事件文件应该包含的内容:

事件文件

两层的办公建筑物

电梯

人

楼门

方向——上升和下降

楼层被占据

人在电梯中

人在某一楼层

某人的目的楼层

人进入电梯

电梯关门

3. 按照类别将事件分组,这样可以帮助读者在练习 1 中正确地标识不同的类。可以采用这样一种大纲格式:每个类列在页面的最左边,同它相关的事件缩进一个制表符后列在这个类的下面一行。存在一个事件对应几个类或者几个事件对应一个类的情况。注意,一些像“方向——上升和下降”这样的事件并非清晰地对应某一个类,它也不可能包含在某一类的组中 (在本例中“方向”是指电梯移动的方向)。这个大纲文件将在本练习和下面的几个练习中得到应用。
4. 现在,将每个类对应的事件分成两组。第一组标识为“属性”,第二组标识为“其他事件”。到目前为止,动作 (行为) 应该包含在“其他事件”组中。当读者将一个动作放到“其他事件”这一组时,应该考虑在属性组中加入相应的一项。例如,事件“楼层被占据”是楼层的一个属性。如果进行更详细地区分,那么在任意时刻,楼层应该被占据着 (由一个人——每一楼层的最大容量) 或者是空的。电梯具有的一些属性包括:是“正在移动”还是“停止”;搭乘或没有搭乘一个乘客;如果它正在移动,其方向是“上升”还是“下降”。楼层按钮的一个属性是它的状态是“开”还是“关”。一个人的属性包括他的目的楼层等。

## 注意

1. 在列出类的属性时,从问题陈述中提到的开始,然后列出由问题陈述直接隐含的那些属性。

2. 如果明显需要某个属性，就将它增加到列表中。
3. 系统设计并非一个理想状态下的完美过程，因此只能尽最大努力实现。当读者在下面的练习中继续完成这个项目时，要准备好不断地修改设计方案。
4. 一个类的对象（确切地说——指向一个类的对象的引用）可能是另一个类的属性，这就是所谓的“类的复合”。例如，在电梯类中包括标识楼层一和楼层二的按钮对象的引用——用来表示某个人在选择目的楼层时按下了哪一层的按钮。从本练习的目的来说，可以同等对待所有的对象——不允许它们采用复合技术。
5. 在第4章中介绍了如何生成随机数。在这个电梯模拟程序中，每当“创建”一个人，就可以使用下列语句：

```
int arrivalFloor = (int)(1+Math.random() * 2);
```

来随机决定这个人出现的楼层。

## D.5 电梯实验室练习3

（预备知识：第6章）

在前面两个练习中，我们体验了采用面向对象的方法对电梯模拟程序进行设计的前两个阶段，即标识用来实现电梯模拟程序的类和标识这些类的属性。在本练习中，我们关注如何确定为实现这个程序所需的类的行为。在下面的部分中，我们将讨论类之间的通信关系。

让我们首先看看现实世界中一些对象的行为。一台收音机的行为包括设置它收听的电台，一辆汽车的行为包括加速（通过踩下油门来实现）、减速（通过踩下刹车来实现）以及改变方向（使用方向盘）。

对象通常不会自动执行它们的行为。但是在调用一个对象的某一方法时，则将执行相应的行为。

1. 继续完成在练习2中创建的事件文件，现在我们将与每个类相关的事件分成了两组——“属性”和“其他事件”。
2. 对于每个对象，增加称为“行为”的第三个组。将激活每个类的对象来完成某件事情（也就是向该对象发送一个方法调用）所对应的行为列在这个组中。例如，一个按钮可以被某人按下，因此将 `pressButton` 作为行为列在 `Button` 类中。这个类的属性（例如一个 `Button` 对象是“开”还是“关”）作为 `Button` 类的实例变量。一个类的方法通常用来处理它的实例变量（例如 `pressButton` 把 `Button` 类的一个属性变为“开”），一个类中的方法常常调用其他类对象的某个方法（例如一个 `Button` 对象调用 `Elevator` 对象的方法 `comeGetMe`）。假设电梯拥有这样的“热”键，当某人按下时按钮就会亮；当电梯到达某一楼层时，它应该激活一个方法 `resetButton`，从而关掉按钮上的灯。电梯应该能够判断出是否按下了一个特定的按钮，因此我们可以提供另一个称为 `isButtonOn` 的行为来判断一个按钮的状态，并返回 `true` 或 `false` 来分别标明按钮当前是“开”还是“关”。另外，还可以采用 `openDoor` 方法和 `closeDoor` 方法来分别对应电梯门的不同操作。
3. 对于赋给一个对象的每个行为，给出关于这个行为作用的一个简单描述。列出这个行为可能改变的属性，同时列出它调用的其他对象的方法。

## D.6 电梯实验室练习 4

(预备知识: 第 6 章)

这是你在开始编写电梯模拟程序之前需要完成的最后一项初始设计。在本节中,我们关心对象之间的关系。本节能够帮助用户“把所有的东西连在一起”,这里可能需要在类的列表、它们的属性以及它们的行为中增加一些东西。

我们已经知道对象不会自动完成某项动作。但是,对象可以通过方法调用的方式来产生回应。

让我们关注电梯模拟程序中类关系的几个例子。在问题陈述中提到“某人按下楼层的呼叫按钮”。这一线索的“主语”是人,而客体为按钮。这是一个有关对象关系的例子,人的对象将向按钮对象发送一个 `pressButton` 的方法调用。在上一个练习中,我们将 `pressButton` 列为 `Button` 类的一个方法。

从某种角度来看,我们留在每个类的“其他事件”一项中的内容就是有关类之间关系的描述。请看下面的陈述:

“某人等待电梯门打开”

在上一个练习中,我们列出了关于电梯门的两个行为,它们分别是 `openDoor` 和 `closeDoor`。现在要判断哪个类发出这些方法调用。在这个用引号括起的陈述中无法得到清晰的结论,因此需要好好考虑一下,然后我们就能够认识到正是电梯自己将这些消息发送给电梯门。这些类之间的关系隐含在问题陈述中,并且同那些明显可以得到的关系相比,将这种关系总结出来更加困难。

现在继续分析电梯模拟程序中每个类的“其他事件”一项中的内容。这一项现在应该包含类之间的关系。这些关系可以总结如下:

1. 发送消息的类
2. 发送的是什么消息
3. 接受消息的类

在每个类的下面,加上“发送给其他类的消息”一项,然后列出类之间剩余的关系。例如在 `Person` 类之下,包含这一项:

`Person` 发送 `pressButton` 方法的调用给 `Button` 类

对于 `Button` 类来说,在“发送给其他类的消息”一栏中放入下面的信息:

`Button` 发送 `comeGetMe` 消息给 `Elevator`

从发送消息的类的角度看来,发送给另一个类的方法调用是一种协作 (*collaboration*) 过程。从接受方的类看来,一个方法的调用激活了一个行为。

在完成这些内容的同时,可以在类中增加其他的属性和行为,这完全是正常的。当完成这个任务之后,将得到一个相当完整的、用来实现电梯模拟程序的类的列表。对于每一个类来说,可以得到这个类的一个相当完备的属性列表、行为列表以及这个类调用其他类的方法的列表。

## D.7 电梯实验室练习 5

(预备知识: 第 6 章)

在前面的练习中,我们介绍了面向对象设计的基本知识,并且通过一个基本的面向对象设计来



展示如何设计电梯模拟程序。从某种角度来说,现在可以开始编写模拟程序了。

1. 对于在前面练习中已经标识好的每一个类,编写出相应的类定义,每一个类定义应该写在单独的、以.java结尾的文件中。
2. 编写一个用来测试这些类以及运行整个电梯模拟程序的驱动程序。注意:在创建模拟程序的一个可以工作的版本之前,首先需要完成下面的练习,因此请使用现在已有的知识来完成电梯模拟程序中可能完成的部分。在下面的练习中,我们将讨论复合类,即创建那些将指向其他对象的引用作为实例变量的类。例如,这个技术可能有助于读者在电梯类中将按钮对象作为电梯的成员。
3. 在模拟程序中,设计一个简单的文本方式的输出,但是它能够全面地反映发生的每一个事件。程序输出的信息可以包括下面这样的字符串:“第一个人到达一楼”,“第一个人按下一楼的呼叫按钮”,“电梯移向一楼”,等等。建议对应程序中相应对象的名词使用大写形式。注意,可以将这一部分推迟到下一个练习完成之后。

## D.8 电梯实验室练习6

(预备知识:第6章)

在上一个练习中已经开始编写模拟程序的代码了。在本练习中将讨论类的复合技术,它可以用创建将其他对象作为成员的类。复合使得我们能创建出这样一个建筑类,它包括指向电梯对象、楼层对象的引用;也可以创建包含指向按钮对象的引用的电梯类。

1. 每当某一个人进入模拟程序时,应该使用new创建一个新的Person对象来代表此人。注意,new应该同--个构造函数的调用一起使用,以便正确初始化这个对象。
2. 列出在模拟程序中实现的类之间的复合关系。修改在上一个练习中创建的类定义,使得它们可以正确反映这些类的复合关系。
3. 完成一个可工作的模拟程序,在下一个练习中将建议在程序中增加或修改问题陈述的内容。

## D.9 电梯实验室练习7

(预备知识:第9章~第11章)

到目前为止,模拟程序的第一个版本已经完成了。在下面几个练习中,我们将修改这个模拟程序,使得它看起来更加真实:这就要求模拟程序中的电梯在不同楼层之间移动时要耗费一定的时间,允许不止一个人同时出现在模拟程序中,采用图形界面表述电梯和其他对象,模拟像电梯或人这样的对象的移动。这个练习更加关注如何增加电梯模拟程序的图形用户界面。图D.1给出了本练习中一个基本的GUI设计。

1. 创建另一个按钮并将它加入到applet中。将一个按钮命名为“楼层一”而另一个命名为“楼层二”。这些按钮用来创建出现在相应楼层的等待电梯的人。
2. 创建一个新的称为FloorCanvas的类,它从Canvas类派生而来。FloorCanvas类的方法paint可以画出一条代表楼层的线条。应该为每一楼层创建一个FloorCanvas对象。
3. 创建一个新的称为ElevatorCanvas的类,它从Canvas类派生而来。ElevatorCanvas类的方法paint可以画出一个实心矩形代表电梯。

4. 创建一个新的称为 Timer 的类，它从 Canvas 类派生而来。Timer 类的方法 paint 可以画出一个圆代表一个时钟。这个时钟应该有一条表示时间的直线。

### 注意

1. 尽力完成图 D.1 所示的图形用户界面组件。
2. 目前，还不需要担心如何将 GUI 同现有代码完成的功能结合在一起，我们将在下面的任务中讨论。
3. 当本练习完成后，除了显示上述的 GUI 组件之外，电梯模拟程序应该能完成原有的一些功能。

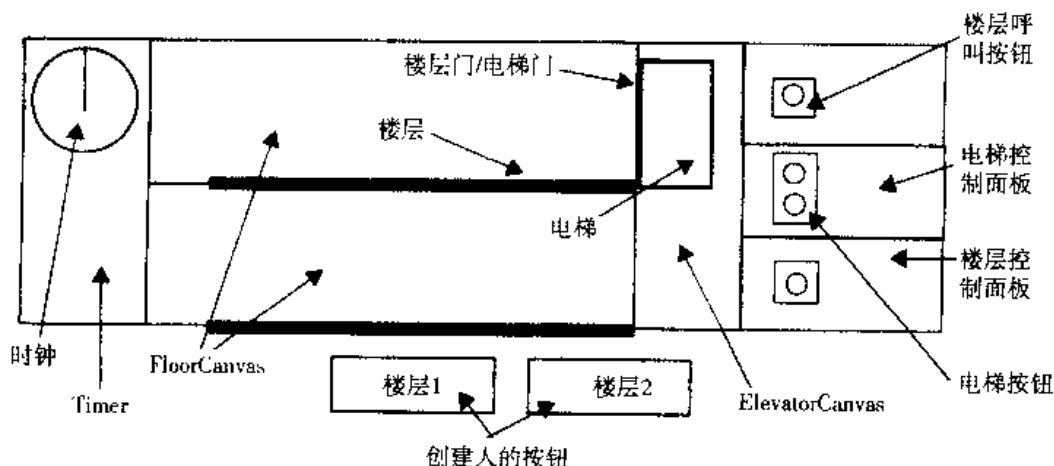


图 D.1 电梯模拟程序 GUI 的图形表示

## D.10 电梯实验室练习 8

(预备知识：第 9 章 ~ 第 11 章)

本练习继续创建练习 7 中开发出的 GUI。在本练习中，需要把模拟程序中发生的事件采用图形的形式加入到 GUI 中。

1. 创建一个称为 ElevatorControlBoard 的新类，它从 Canvas 类派生而来。ElevatorControlBoard 类的 paint 方法可以画出一个包含两个圆的矩形，每个圆代表电梯中的一个按钮。
2. 创建一个称为 FloorControlBoard 的新类，它从 Canvas 类派生而来。FloorControlBoard 类的 paint 方法可以画出一个包含一个圆的矩形，这个圆代表楼层的呼叫按钮。需要初始化两个 FloorControlBoard 对象。
3. 写出可以表示文本输出和按钮之间可视化交互关系的规则。也就是说，每当在模拟程序中按下一个按钮时，按钮应该发亮。例如，如果一个人楼层一按下了一个呼叫按钮，那么 FloorControlBoard 画板应该将代表这个按钮的圆变成黄色。

## D.11 电梯实验室练习 9

(预备知识：第 13 章)

本练习包括提供 GUI 和模拟程序发生的事件之间交互关系的必要步骤。大多数情况下这些交互

将采用多线程实现。

1. 修改模拟程序使得同时可以在建筑物中出现多个人。记住，在同一时刻最多有一个人在电梯中，最多每层出现一个人；因此在模拟程序中可能同时出现0、1、2、3个人。当按下相应的按钮时，将有一个新人进入模拟程序（假设目的楼层没被其他人占据）。每个人应该表示为一个线程。
2. 将电梯修改为一个独立的线程，电梯应该不断查看每层的呼叫按钮。
3. 将时钟修改为一个独立的线程，电梯在两个楼层之间的移动应该耗费时钟走一圈的时间（即时钟的指针旋转360度），并且时钟仅对电梯有效。提供一个文本输出来指明何时时钟走完一圈，一圈的时间大约可以调用60次Timer类的paint方法。

#### 注意

1. 电梯现在变得“聪明了”，它“知道”何时移动。
2. 程序仍然应该在控制台上输出信息，这可以帮助读者了解程序是否运行正常。
3. 电梯在练习中移动时没有相应的图形表示，在下一个练习中将要求实现电梯移动的动画。
4. 时钟的指针现在还不能移动，在下面的练习中将要求模拟时钟指针移动的动画。

## D.12 电梯实验室练习 10

（预备知识：第13章）

本练习包含在程序中实现动画的几个必要步骤。

1. 修改模拟程序，使得时钟可以走动。每当电梯开始在楼层之间移动时，时钟应该也开始走动。当时钟的指针到达它的初始位置时，电梯应该到达目的楼层。
2. 修改程序，使得电梯在楼层之间移动时有相应的图形表现。每当时钟的指针开始移动时，代表电梯的“矩形”也应该在楼层之间移动，当时钟指针走完一圈时，电梯应该停在目的楼层。
3. 修改模拟程序，使得楼门和电梯门在开关的时候都有相应的动画。

#### 注意

1. 楼门和电梯门在同一层的开关动作应该一致，因此使用一条线表示即可。

## D.13 电梯实验室练习 11

（预备知识：第14章）

本练习包含想要在程序中添加其他的动画及声音的几个必要步骤。图D.2中显示出了GUI中动画对象的放置方法。

1. 修改模拟程序，使得它可以显示一个在楼层中行走的人，可以使用GIF文件；同样可以使用GIF文件动画来显示一个人离开楼层的情况。
2. 修改模拟程序，使得它可以显示一个人在电梯中，可以使用与步骤1中相同的GIF文件。
3. 在FloorControlBoard中添加一个GIF图像来表示楼层中的灯。每当电梯到达某一楼层时，这

- 个 GIF 图像应该“发亮”，然后在电梯离开时熄灭。
- 4. 在 ElevatorControlBoard 中增加一个 GIF 图像用来表示一个电铃。
- 5. 每当有人按下一个按钮时，增加一个“点击”按钮的声音。
- 6. 每当有人在楼层中走动时增加一个“脚步”的声音。
- 7. 当电梯在楼层中移动时增加一段“优雅”的音乐（当然摇滚乐也行！）。
- 8. 当响铃时增加一段音乐。

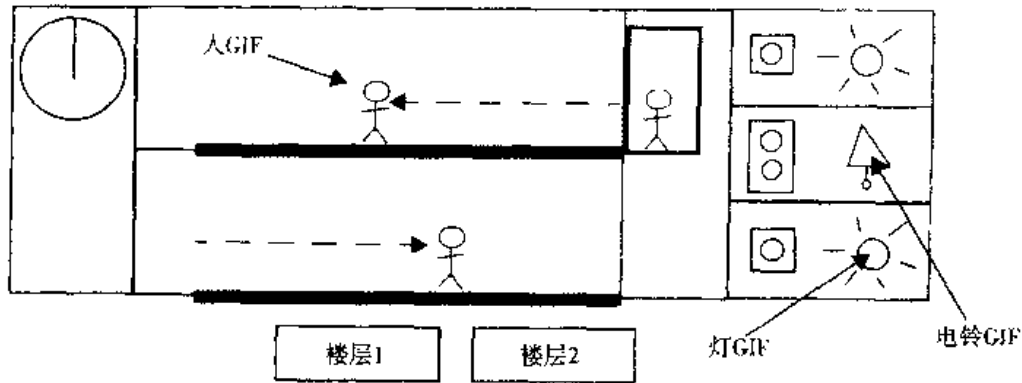


图 D.2 电梯模拟程序 GUI 的图形表示

#### 注意

- 1. 可能需要两个 GIF 图像来表示楼中的灯。一个图像用来表示灯亮的情形，一个图像用来表示灯熄灭时的情形。
- 2. 可能需要为用户提供一个“隔音”选项，它类似于“响铃”，是一种特殊的声音，因为同时播放太多的声音对用户来说可能会感到不愉快。

## D.14 电梯实验室练习 12

（预备知识：第 14 章）

本练习包括上述练习的动画制作。

- 1. 使用不同的 GIF 图像在模拟程序中代表不同的人。

#### 注意

- 1. 在 Internet 上的各个服务器中搜寻图像文件和声音文件；读者可以在模拟程序中使用这些文件。
- 2. 为了使我们的模拟程序更具有真实感，应该随机地使用表示人的 GIF 文件。

#### 问题

- 1. 应该怎样使电梯可以处理期望的交通流量？
- 2. 如果模拟一个 3 层的建筑，为何问题会变得更加复杂？
- 3. 一旦我们能够创建一个电梯对象，当然就可以创建更多的对象。如果有几部电梯，将会产生什么样的问题？当有人请求搭乘电梯的时候，如何决定由哪部电梯为他服务？

4. 在理想状态下, 仅仅允许每部电梯搭乘1个人。如果将限制放宽到3个人将会发生什么问题?

## D.15 建议的修改

1. (预备知识: 第17章) 修改程序, 使电梯可以最多允许3个人在同一楼层等待。每一楼层应该拥有一个 Queue 类的对象。
2. (预备知识: 第14章) 使用一系列的 GIF 图像来表示一个人“走过”楼层(即我们应该看到人的腿部和手臂的动作)。
3. 将建筑物的楼层由2层改为3层。
4. 完成上述修改后, 在模拟程序中增加一部或多部电梯。
5. 完成上述两个修改之后, 修改每部电梯服务的楼层限制。将其中的一部电梯设置为“快速”电梯, 使其从底层直达顶层而不在中间任一层停留。
6. 将每个楼层中的等待人数增加到6个或更多, 同时将一部电梯可以搭乘的人数增加到3个或更多。
7. 修改电梯模拟程序, 使它具有客户/服务器的性能, 例如当用户点击某个楼层的按钮时, 模拟程序将链接到“人员服务器”并询问有关此人的信息(即此人的相关号码以及必要的 GIF 图像)。
8. 修改模拟程序, 使用 Observable 类和 Observer 接口, 建议可以设置一个人员来“观察”电梯。注意: 在本应用程序中不需要用到前面的修改。
9. 修改模拟程序, 使它可以定时(例如每分钟)将“活动状态”写入文件。状态信息应包括建筑物中的全部人数、每一楼层等待的人数、电梯中的人数等。

## 参考文献

- (Ai96) Aitken, G., "Moving from C++ to Java," *Dr. Dobb's Journal*, March 1996, pp. 52-56.
- (An96) Anuff, E., *Java Source Book*, New York, NY: John Wiley & Sons, Inc., 1996.
- (Ar96) Arnold, K., and J. Gosling, *The Java Programming Language*, Reading, MA: Addison-Wesley Publishing Company, 1996.
- (Ba96) Bandarpalle, R., and R. Ratnakar, "Distributed Business Applications Using Java: An Implementation Framework," *Java Report*, July/August, 1996. pp. 46-50, 64.
- (Be96) Berg, C., "How Do I Send e-mail from a Java Applet?" *Dr. Dobb's Journal*, August 1996, pp. 111-113.
- (Bo96) Boone, B., "Multitasking in Java," *Java Report*, May/June 1996, pp. 27-33.
- (Cg96) Cargill, T., "An Overview of Java for C++ Programmers," *C++ Report*, February 1996, pp. 46-50.
- (Ca96) Caron, J., "Java: A Status Report," *Web Techniques*, June 1996, pp. 30-37.
- (Ch96) Chaffee, A. D., "Enhancing the Web," *Java Report*, March/April 1996, pp. 24-26.
- (Co96) Cornell, G., and C. S. Horstmann, *Core Java*, The SunSoft Press, Upper Saddle River, NJ: Prentice Hall, 1996.
- (Dc96) Daconta, M. C., *Java for C/C++ Programmers*, New York, NY: John Wiley & Sons, Inc., 1996.
- (Da96) Danesh, A., *Teach Yourself JavaScript in a Week*, Indianapolis, IN: Sams.net Publishing, 1996.
- (De95) December, J., *Presenting Java*, Indianapolis, IN: Sams.net Publishing, 1996.
- (Fl96) Flanagan, D., *Java in a Nutshell*, Sebastopol, CA: O'Reilly & Associates, Inc., 1996.

- (Fr96) Freeman, A., and D. Ince, *Active Java*, Reading, MA: Addison-Wesley Publishing Company, 1996.
- (Fr96a) Freeman, A., and D. Ince, "The java.net Library," *Java Report*, May/June 1996, pp. 35-41.
- (Fi96) Fitzgerald, P., "Becoming a JavaScript Author," *Java Report*, May/June 1996, pp. 55-56.
- (Fl96) Flynn, J., and B. Clarke, "The World Wakes up to Java!" *Computer Technology Review*, 1996, pp. 33-37.
- (Fl96a) Flynn J., and B. Clarke, "How Java Makes Network-Centric Computing REAL," *Datamation*, March 1, 1996, pp. 42-43.
- (Ga96) Gabler, C., "Java: Adding Interactivity in HTML Pages," *Web Techniques*, April 1996, pp. 57-61.
- (Go96) Goodman, D., *JavaScript Handbook*, Foster City, CA: IDG Books, 1996.
- (Go96) Gosling, J.; F. Yellin; and The Java Team, *The Java Application Programming Interface*, Volume 1: Core Packages, Reading, MA: Addison-Wesley Publishing Company, 1996.
- (Gr96) Greenbaum, J., "Java Fever!" *Software Magazine*, May 1996, pp. 36-44.
- (Gl96) Gulbransen, D., and K. Rawlings, *Creating Web Applets with Java*, Indianapolis, IN: Sams.net Publishing, 1996.
- (Gu96) Gurewich, N. and O. Gurewich, *Java Manual of Style*, Emeryville, CA: Ziff-Davis Press, 1996.
- (Ha96) Harms, D., et. al., *Web Site Programming with Java*, New York, NY: McGraw-Hill, Inc., 1996.
- (Hy96) Hayashi, A. M., "Reality Behind the Java Hype," *Application Development Trends*, February 1996, pp. 78-85.
- (He96) Hemrajani, A., "Examining Symantec's Cafe," *Dr. Dobb's Journal*, August 1996, pp. 78-82.
- (Ho96) Hof, R. D., and J. Verity, "Scott McNealy's Rising Sun," Cover Story, *Business Week*, January 22, 1996, pp. 66-73.
- (Ja96) Jackson, J. R., and A. L. McClellan, *Java by Example*, The SunSoft Press, Upper Saddle River, NJ: Prentice Hall, 1996.
- (Jo96) Johnston, S. J., and J. Swenson, "Microsoft Buys a Cup of Java," *Information Week*, April 29, 1996, pp. 14-16.
- (Jv96) Jovin, D., "Developing Database-Driven Applications in Java," *Java Report*, July/August, 1996, pp.53-56.
- (Ko96) Korzenowski, P., "Java: Cross-Platform OO Language for Distributed Development," *Application Development Trends: Advertising Supplement*, February 1996, pp. S-15-S-21.
- (La96) Lalani, S., and K. Jamsa, *Java Programmer's Library*, Las Vegas, NV: Jamsa Press, 1996.
- (Le96) Lemay, L., and C. L. Perkins, *Teach Yourself Java in 21 Days*, Indianapolis, IN: Sams.net Publishing, 1996.
- (Le96a) Lemay, L., and C. Perkins, "Yes, Java's Secure. Here's Why," *Datamation*, March 1, 1996, pp. 47-49.
- (Lo96) Lorenzo, J., "Exploring Java's Animator Applet," *Web Developer*, Spring 1996, pp. 9-10.

- (Me96) Meese, P. D., "The Java Tutor: Creating Your Own First APP," *Java Report*, March/April 1996, pp. 37-46.
- (Me96a) Meese, P. D., "The Java Tutor," *Java Report*, May/June 1996, pp. 43-48.
- (Me96b) Meese, P. D., "Improving Marquees with Double Buffering," *Java Report*, July/August, 1996, pp. 35-39.
- (Me96a) Meese, P. D., "The One Hour Java Applet," *Datamation*, March 1, 1996, pp. 51-61.
- (My96) Meyer, M., "True, Online Multimedia," *Java Report*, July/August, 1996, pp. 58-59.
- (Na96) Naughton, P., *The Java Handbook*, Berkeley, CA: Osborne-McGraw Hill, 1996.
- (Ne96) Newman, A., et. al., *Using Java: Special Edition*, Indianapolis, IN: Que Corporation, 1996.
- (Ni96) Niemeyer, P., and J. Peck, *Exploring Java*, Sebastopol, CA: O'Reilly & Associates, Inc., 1996.
- (Oa96) Oaks, S., "How Do I Create My Own UI Component?" *Java Report*, March/April 1996, pp. 64, 63.
- (Oa96a) Oaks, S., "Two Techniques for Handling Events," July/August, 1996, p. 80.
- (Pr96) Perkins, C. L., "The Big Picture," *Java Report*, March/April 1996, pp. 47-53, 56.
- (Pe96) Pew, J. A., *Instant Java*, The SunSoft Press, Upper Saddle River, NJ: Prentice Hall, 1996.
- (Re96) Reynolds, M. C., "Java Programming from the Grounds Up," *Web Developer*, Spring 1996, pp. 30-40.
- (Ro96) Rodley, J., "Java Applets and Netscape," *Web Techniques*, April 1996, pp. 70-72.
- (Sn96) Sams.net Publishing, *Java Unleashed*, Indianapolis, IN: Sams.net Publishing, 1996.
- (Sr96) Sarna, D. E., and G. J. Febish, "Java's Place in the Wide OLE World," *Datamation*, February 1, 1996, pp. 25-27.
- (Sv96) Savetz, K. M., "Jiving with Javascript," *Web Developer*, May/June 1996, pp. 58-59.
- (Sc96) Scotkin, J., "Three Tiers for Java," *Java Report*, May/June 1996, pp. 59-62.
- (Sc96a) Scotkin, J., "Business Uses for Java," *Datamation*, March 1, 1996, pp. 40-41.
- (Sc96b) Scotkin, J., "Java in the Corporate Environment," *Java Report*, March/April 1996, pp. 54-56.
- (Sc96c) Scotkin, J., "Myths, Reality, and When to Use Java," July/August, 1996, pp. 74-77.
- (So96) Scott, A., "VRML and JavaScript," *Java Report*, July/August, 1996, pp. 41-45.
- (Se96) Semich, B., and D. Fisco, "Java: Internet Toy or Enterprise Tool?" *Datamation*, March 1, 1996, pp. 28-37.
- (Si96) Singleton, A., "Wired on the Web," *BYTE*, January 1996, p. 77-80.
- (Su96) Sun Microsystems, Inc., *Introduction to Java Programming*, Course SL-230, Sun Educational Services SunService Division, Sun Microsystems, Inc., MS UMIL07-14, Part Number 802-6281-02, Revision B, March 1996.
- (Su96a) Sun Microsystems, Inc., *Java Application Programming*, Course SL-270, Sun Educational Services SunService Division, Sun Microsystems, Inc., MS UMIL07-14, Part Number 802-6282-02, Revision B, March 1996.



- (Te96) Tessier, T., "Using JavaScript to Create Interactive Web Pages," *Dr. Dobbs's Journal*, March 1996, pp. 84-96.
- (Ti95) Tillel, E., and M. Gaither, *60 Minute Guide to Java*. Foster City, CA: IDG Books, 1995.
- (Tr96) Tropeano, D., "Getting into OT," *Java Report*, March/April 1996, pp. 27-29, 46.
- (Tr96a) Tropeano, D., "Class Design," *Java Report*, July/August 1996, pp. 31-34.
- (Ty96) Tyma, P., "Tuning Java Performance," *Dr. Dobbs's Journal*, April 1996, pp. 52, 55-57, 90-92.
- (Vd96) Valdes, R., "Methods for Motion: Revisiting Animation on Web Pages," *Web Techniques*, June 1996, pp. 45-49.
- (Vh96) van Hoff, A.; S. Shaio; and O. Starbuck, *Hooked on Java*. Reading, MA: Addison Wesley Publishing Company, 1996.
- (Vh96a) van Hoff, A. S., S. Shaio, and O. Starbuck, "What is this Thing Called Java?" *Datamation*, March 1, 1996, pp. 45-46.
- (Vh96b) van Hoff, A., "Believing the Hype," *Java Report*, March/April 1996, pp. 31-34.
- (Vl96) van der Linden, P., *Just Java*, The SunSoft Press, Upper Saddle River, NJ: Prentice Hall, 1996.
- (Vn96) Venditto, G., "Java: It's Hot, But Is It Ready to Serve?" *Internet World*, February 1996, pp. 78-80.
- (Ve96) Vermeulen, A., "An Asynchronous Design Pattern," *Dr. Dobbs's Journal*, June 1996, pp. 42-44.
- (Wl96) Waldo, J., "Programming with Java," *UNIX Review*, May 1996, pp. 31-37.
- (Wa96) Walter, S. J., and A. Weiss, *The Complete Idiot's Guide to JavaScript*, Indianapolis, IN: Que Corporation, 1996.
- (Wi96) Wilson, A., "Porting Windows Applications to Java: Part I," *Web Techniques*, June 1996, pp. 39-43.
- (Wo96) Wong, H., "A Look at Layout Managers," *Java Report*, May/June 1996, pp. 49-51.
- (Wo96a) Wong, H., "Writing Your Own Layout Managers," July/August, 1996, pp. 65-73.